M I N T

Machine Independent Organic Software Tools

M.D. Godfrey, D.F. Hendry
H.J. Hermans, R.K. Hessenberg

Contents pg. xv

Online Version Revision History

Created on 27 October 2016.
Using pdftex: 3.14159265-2.6-1.40.17 (TeX Live 2016)

January 2001:

Initial TEX version created by conversion, using the MINT system, of the COMADS source files which had originally been used to produce publication-ready images for Academic Press.

28 January 2002:

Partial corrections, mainly from Barrie Stott. The corrections are fairly complete through Chapter 2. Chapter 4 has also been extensively rearranged.

16 February 2002:

The optional TRAP instruction has been re-implemented and corrected. Functions that use TRAP, T$ and TRMIX for example, have been updated.

The corrections to the text provided by Barrie have been applied, and additional corrections were made to Chapters 13, 14 and 15.

29 March 2002:

The VM has been changed to accept a string argument. This string becomes the first string read by MINT, before reading input from the normal input source. Thus, mint -i "SI startup.mnt" will cause the file startup.mnt to be read by the SI directive. See Section 15.3.1.

26 December 2002:

Compiling on an *ibook* reminded me that PDUMP format is "endedness" sensitive. This is now made clearer in Chapter 14 and both a big-end and a little-end PDUMP file of the compiler are included in the system files.

21 May 2003:

The reference C-coded VM has been changed to use the GNU *readline* library including the save and restore of the history file. This makes interactive use much more convenient. This feature can be turned off by means of a directive for systems that do not provide the readline library.

14 January 2004:

An Appendix has been added to the book. The Appendix provides a chronology of significant corrections and changes to the compiler and to the C-coded virtual machine.

15 August 2004:

Changes were made to Section 1.9 to reflect the resources used by the current system and by the Linux reference implementation.

11 July 2006

Minor updates to clarify and document recent changes. Table 8-1 is now an up-to-date summary of compiler directives.

19 September 2006:

Changes to the VM were made to provide correct operation on x86_64 systems. Currently, the system is still 32-bit, but it executes correctly when compiled on 64-bit systems.

The requirement for separate PDUMP files for Little-ended and Big-ended systems has been eliminated. The system type and PDUMP file type are tested and the byte order of integers is reversed if necessary.

16 April 2007:

Minor typographical changes, corrected the Index, and entered hyperlinks; one on the front page pointing to the Contents, and in the Contents pointing to each Chapter.

28 July 2007:

Minor typographical changes mostly due to use of *eplain* and *Metapost*. Hyperlinks were introduced, and the index (using eplain functions) was improved.

Preface to the Third Edition

This Edition describes MINT3. MINT3 includes a substantial number of new features beyond the MINT system described in the book *MINT, Revised Second Edition* (1985). The most significant changes are the introduction of dictionaries by means of the CLASS DICT, inclusion of priority (shunt factor) in the dictionary record (rather than the CLASS record), and the inclusion of B-tree functions. The system has been changed to use 32-bit words and to use a large (typically 2^{24} words) virtual address space. Strings are now stored 4 characters per word.

The new dictionary mechanism is used to improve auto-compilation of the system. Auto-compilation is now much simpler, more flexible, and more clearly realized within the standard MINT facilities. One specific improvement is that the layout and contents of dictionary records may be changed during auto-compilation in a relatively convenient manner. This will facilitate any further extensions of the system.

These changes and additions have resulted in a reduction of the listing length of the compiler, including the auto-compilation procedures and the B-tree functions, from 58 pages to 55 pages.

The new C-coded VM makes use of an environment variable. The VM loads the compiler in PDM format (MCOMP.PDM) from the directory pointed to by the variable MINT_HOME.

There are two other detail changes to the MINT compiler, and a considerable number of corrections and clarifications in the text. The changes to the compiler are that an identifier may be of any length (no longer restricted to up to eight characters), and the OPINT and OPINTD operators were rewritten so that the sign is always correctly interpreted and the correct interpretation of integers of magnitude greater than 64K is made easier. The value of the first change should be obvious. Its implementation required making the dictionary records variable length. The operators which manage variable-length records are also available, and are described in Chapter 9. The second change came about due to the increasing use of

MINT for applications where the use of 16 bits for the size of data-storage units is inadequate. In the current implementations the data-storage unit is 32 or more bits and additional data space is allocated beyond the MAXPLOC value. This extension has permitted MINT's use on problems which involve very large amounts of data. Generally, in these situations some form of paging of VSTORE is desirable in order to avoid large real memory requirements. As part of these changes, the C Virtual Machine implementation has become the standard. The Chapters on The Apple-II and Sperry UNIVAC implementations have been replaced by Chapter 15 – the C Implementation. The previous Chapters 15 and 16 can be made available for those interested in history.

One incompatibility has been introduced by these changes: the CLASS directive now requires three fields instead of four. The precedence field is no longer used. All instances of CLASS should simply delete the precedence field. If non-standard precedence is required, use the PRIORITY directive (See Section 4.9.2). No other incompatibilities are known at present. The current compiler is identified as Version 3.0. The most significant correction to the text is the replacement of Figure 9-2 and the accompanying text which describe the layout of record lists. The descriptions of the EMULATE and TRAP directives have been clarified.

The diagnostics have been further improved and extended based on the discovery of a few additional errors in Virtual Machine implementations which got through undetected. Current experience is that successful operation of the diagnostics is practically certain to imply correct operation of the Virtual Machine. The only significant Virtual Machine mechanisms which are not checked by the diagnostics are use of nonzero segment numbers, and the operation of EMULATE and TRAP. The compiler will operate without implementation of these features, so they may be verified after the compiler is fully operational.

The DO operator definition was extended to allow its operation on primitives. This is a simple change in the VM definition which improves uniformity.

Several new Virtual Machine implementations are now in use including the implementation written in C. The C implementation is now the standard for practically all systems.

The most ambitious MINT-written system of which we are aware is a VLSI design system. The system provides a very compact notation for logic definition, and provides flexible multi-level simulation. Starting in the late 1970's, this system was used for a number of VLSI designs, including a chip-set for a mainframe architecture. In the case of the mainframe chip-set, the design system was used to define and simulate the entire chip-set, which

was composed of about 1 million transistors. This became the UNISYS 2200 Series product in the 1980's.

For this edition the source text of the Second Edition, in Sperry Univac COMADS format, was converted to TEX. This was done "semi-automatically" using a MINT program. This conversion permits production of the text in PostScript and PDF format thus making it accessible on the WEB.

Since this edition is available on the WEB, the compiler listing is not included as an Appendix. Instead, it is available along with all the source code at: mint3-dst.tgz

Michael D. Godfrey

May 2021

Preface to the Second Edition

This Edition is different from the first edition in two substantial ways. First, several corrections have been made. The most significant of these is the correction of the operation diagram for DICMATCH which appeared on page 202. The other corrections are all minor, being either typographical or obvious. The second difference is the introduction of several improved or new components. The significant improvements are: the INCH primitive which replaces GETSTR, a new virtual memory arrangement which permits use of 64K words of virtual memory and permits more efficient allocation of object text, a new portable format which is more compact and which is checksum and sequence checked, consolidation of the OPENxF primitives into the one new primitive OPENF, and provision of more powerful and selective diagnostics at the virtual machine level. These changes will not cause substantial incompatibility with respect to current source text. Where appropriate, routines are provided which allow continued operation of old constructs, such as a *function* GETSTR which uses INCH. In other cases, obvious changes should be made in source text which is based on the First Edition.

Chapter 12 has been expanded to include both MINT techniques and examples. Chapter 15 now contains a description of the MINT implementation on an Apple-II system instead of the Intel 8080 implementation.

During the period since the first edition we have benefited substantially from discussions with and contributions from R. N. Riess. In addition, he contributed the EMULATE primitive which is described in Chapter 6.

This edition has been produced by the Sperry Univac COMADS system, as was the first edition. Thus, the process of production of the new edition was to write the new text, edit it into the first edition files, apply the usual spelling checking and analysis tests, run proof copy, correct and run final camera-ready copy. It is again a pleasure to acknowledge the help of Richard H. Acquard who is responsible for the COMADS language processor. In addition, Paul J. Pontinen, who has responsibility for the implementation of COMADS on the COMp 80 microfilm processor, has been

particularly helpful in providing additional processing capabilities. These enhancements have improved the appearance of the result, and the ease of its production.

We have been pleased by the reactions of readers during the two years since the publication of the first edition. There have been numerous requests for copies of the system. We have found in practice that most potential users can accept the system on ANSI-format magnetic tape. Due to the problems of formats of cassettes and floppy disks, and our own access to facilities, we have had to restrict availability to magnetic tape, a floppy disk suitable for bootstrap loading into an Apple-II+ system, or, in special cases, transmission over communication lines.

Michael D. Godfrey

June 1982

Preface to the First Edition

The tools described in this monograph are intended to improve the efficiency of computer use and increase the value of the written instructions (termed *software*) which control the operation of computing machines. This is achieved through simplification and generalization of basic constructs, and through separation of the written software from the machines on which the software may operate.

It is intended that this monograph serve several purposes. First, it represents a complete summary of a body of research and development which has been underway since the late 1960's. Second, the content and level of presentation are such that the text may be used for advanced undergraduate or graduate courses in design and implementation of languages, virtual machines, or simple stack based processors. In addition, the text contains information which should be of interest to professional software writers or system designers. The example implementations of the system can be used as trial implementations for study, or may be used as the basis for production application implementations. In practice, these tools have been found to be highly effective for a wide range of applications on machines of widely differing structure. We hope that this monograph will help others to make effective practical use of these tools and techniques.

Until recently these tools were called the SNIBBOL system. While SNIBBOL is as good a name as any other (better than most we could think of), confusion with SNOBOL and other possible misleading associations led us to change the name to MINT (Machine-INdependent Organic Software Tools).

MINT has been put to practical use at several places. This practical use has been essential to the development of the system and, we hope, productive in its own right. The initial development of MINT took place in the early 1970's while D. F. Hendry was at the University of London Institute for Computer Science. MINT was used there as a part of the M.Sc. course. Many further uses have occurred in more recent years. We are aware of MINT implementations for about ten different computer systems.

Many users of MINT are known to the authors. Many of these have contributed significantly to further development of the system. We would like to acknowledge this help even though it is not feasible to list all the individuals who have made such contributions.

D. F. Hendry has been responsible for most of the basic concepts of MINT as it exists today. The current compiler implementation was created by Hendry. Initially R. K. Hessenberg tested and corrected the compiler, as well as contributing helpful insights and improvements. Subsequently, the compiler has been modified and extended by H. J. Hermans and M. D. Godfrey. Hessenberg and Hendry wrote the initial version of the Sperry Univac Series 1100 interpreter (described in Chapter 16) with some help from Godfrey, who has subsequently modified and extended the implementation. Hermans wrote the Intel 8080 interpreter which is described in Chapter 15. An initial MINT manual was prepared by Hendry and Hessenberg. That manual was extensively used in the preparation of Chapters 2 through 11 and Chapter 13 of the present monograph. The completion of the monograph in its present form has been carried out by Godfrey and Hermans.

This entire text, including all Tables and Figures, was prepared by means of a Sperry Univac computer-based documentation system (COMADS). It is hoped that the text reflects the quality of this system. The system greatly facilitated the writing task, as many time-consuming activities, such as proof-reading, were carried out by the computer. The fact that the entire document is stored in the computer has allowed use of the actual source files where language text is given. Thus, all such text has been processed as source text by the MINT system, and therefore checked for correctness. The use of computer-based tools did not completely remove the need for human assistance. Specifically, Richard H. Acquard has been extremely helpful in giving advice and providing support concerning the operation of the COMADS system.

This monograph is unusual in that the complete source text of the system (compiler, virtual machine, syntax analyzer, other text, and examples) are given. This demonstrates the compactness and readability of the system. By agreement with Academic Press Inc. (London) Ltd. the authors retain the copyright of this machine-readable source text.

Copies of the system in machine-readable form may be obtained by writing to me. When such a request is made, it is essential to state the required medium from the following choices:

1. Industry standard magnetic tape, 9-track, 1600 bpi, ASCII coded card images.
2. Standard cassette tape, ASCII coded images.

3. Another recording device which has a standard RS-232C interface. In this case the recipient must provide the recording device and the recording medium.

There will be a charge made in order to cover the cost of copying.

Michael D. Godfrey

May 1980

Contents

Preface to the Third Edition v

Preface to the Second Edition ix

Preface to the First Edition xi

1.0 The MINT System 1

1.1 Introduction 1
1.2 Scope 1
1.3 Purpose 2
1.4 Background 3
1.5 MINT Functional Structure 5
1.6 Organic Programming 6
1.7 The Dictionary System 7
1.8 Uniformity 7
1.9 Compactness 8
1.10 Storage Organization 9
1.11 The Virtual Machine 10
1.12 Introductory Examples 12

2.0 MINT Language Components 15

2.1 Introduction 15
2.2 Definitions 15
2.3 Identifiers 21
2.4 Internal Compiler Identifiers 29
2.5 Constants 29
2.6 Diagnostics 33
2.7 Problems 35

3.0 Program Listing Control 37

3.1 Introduction 37
3.2 Listing Options 37
3.3 Comments and Pagination 38

3.4 The TITLE Directive 38
3.5 Problems 39

4.0 MINT System Structure 41

4.1 Introduction 41
4.2 Basics of the VM(M) Virtual Machine 41
4.3 Compiler Operation 43
4.4 The Dictionary Facilities 43
4.5 Use of NOW and PDUMP 53
4.6 Auto-compilation Facilities 54
4.7 Compiler States and Data Declarations 54
4.8 MINT Expressions 57
4.9 Precedence 60
4.10 Problems 67

5.0 The Macro Facility 69

5.1 Introduction 69
5.2 Macro Bodies 69
5.3 Macro Parameters 69
5.4 MINT System Macros 70
5.5 Some Additional Macros 71
5.6 Problems 72

6.0 Basic MINT Constructs 73

6.1 Introduction 73
6.2 VSTORE Referencing 73
6.3 Operand Stack Management 76
6.4 Control Transfer 78
6.5 Conditional Selection and Iteration 79
6.6 Miscellaneous Constructs 84
6.7 Problems 88

7.0 Functions 91

7.1 Introduction 91
7.2 Identified Functions 91
7.3 Anonymous Functions 96
7.4 Miscellaneous Compiler Functions 99
7.5 Summary of Compiler Functions 100
7.6 Problems 102

8.0 Directives and Immediate Execution 103

8.1 Introduction 103
8.2 Input Parameters for Directives (IPAR) 103
8.3 Referencing Directives as Functions 104
8.4 Immediate Execution 105
8.5 The Class Directive 106
8.6 Miscellaneous Directives 106
8.7 Summary of Compiler Directives 108
8.8 Problems 110

9.0 Lists and Free-Space Management 113

9.1 Introduction 113
9.2 Basic List Structure 113
9.3 Adding to and Removing from a List 114
9.4 Free-Space Management 114
9.5 Item Lists 116
9.6 Record Lists 118
9.7 Variable Length Records 120
9.8 The Dictionary List Structure 121
9.9 B-Tree Data Access 123
9.10 Problems 126

10.0 The External and String Operators 127

10.1 Introduction 127
10.2 MINT String Format 127
10.3 Initialization of External Segments 128
10.4 Input Facilities 129
10.5 Compiler Input Facilities 131
10.6 Output Facilities 135
10.7 Compiler Output Facilities 136
10.8 Closing of Segments 139
10.9 The String Matching Primitives 140
10.10 The COMPILE Function 141
10.11 Problems 141

11.0 The Syntax Analysis System 143

11.1 Introduction 143
11.2 Phrase Structure Analysis 143

11.3 Parsing Functions 146
11.4 Optional Elements 147
11.5 Phrase Function Usage 147
11.6 Listing of M-TRAN 148

12.0 MINT Techniques and Examples 151

12.1 Introduction 151
12.2 Entering Text 151
12.3 Translation and Manipulation of Text 151
12.4 Analysis and Diagnostic Techniques 152
12.5 A Simple Calculator 154
12.6 Text Editing Directives 156
12.7 Instruction Execution Analysis 166

13.0 The VM(M) Virtual Machine 171

13.1 Introduction 171
13.2 The Virtual Machine Architecture 171
13.3 Virtual Machine Object Text Format 173
13.4 Loading the Virtual Machine 174
13.5 The Virtual Machine Instruction Set 177
13.6 Summary of Virtual Machine Primitives 233

14.0 The Distributed MINT System 235

14.1 Introduction 235
14.2 Virtual Machine Diagnostics 236
14.3 The Compiler Object File 238
14.4 Additional Source Text Files 239
14.5 Character Order Reversal in Strings 239
14.6 Compiler Creation and Source Structure 239
14.7 System Generation Sequences 243

15.0 The C Implementation of VM(M) 245

15.1 Introduction 245
15.2 VM Components 245
15.3 Operation of the VM 249

Appendix: History of Corrections and Changes 255

Subject Index 259

The love of economy is the root of all virtue.
G. B. Shaw

1. The MINT System

1.1 Introduction

The MINT system is a set of tools to facilitate communication with, and operation of, computers. These tools provide a high level software environment which is machine-independent and open-ended. The machine-independence implies that the MINT system, and MINT based applications, are readily portable to many machines. The open-endedness implies considerable flexibility in altering or extending the language facilities. The language itself allows sequences and expressions at as high or as low a level as is desired.

MINT is implemented in terms of a *Virtual Machine* which allows exactly the same (virtual) environment to exist regardless of the actual machine on which the system is operating. This Virtual Machine is referred to as the VM(M) Virtual Machine, and the instructions which the Virtual Processor executes are the VM(M) instruction set. Careful definition of this Virtual Machine contributes to the portability, compactness, efficiency, and verifiable correctness of the system.

1.2 Scope

The scope of MINT is very wide both in terms of machines on which it may operate and in terms of potential applications. At present MINT operates on such machines as the Apple-II and on large general-purpose mainframe systems such as the Sperry Univac Series 1100. Applications which have been written entirely in MINT include a number of compilers and assemblers for both small and large machines, language interpreters, a text editor, and interactive dialog systems. These implementations were all relatively low cost in terms of development and implementation effort when compared to similar efforts using conventional techniques. The resulting programs are readable, and portable to any new machine.

In addition to its direct usefulness as a set of development and imple-

mentation tools, MINT can be an effective means of communication. MINT written text is precise, compact, and readable. The MINT Virtual Machine is a simple and carefully structured machine which displays the essential features of a stack-based (or zero-address) machine architecture. Thus, the MINT system provides an effective means of communication between people, between machines, and between people and machines.

The emphasis on effectiveness of communication makes MINT suitable for teaching computing principles and techniques. The system may be used to teach or learn about stack-based architecture, virtual machine design and implementation, compiling, macro structure, parsing, and concepts and techniques of portability. In this text we have not attempted a strict separation of these subjects. This is because we feel that they are not reasonably separable. Much of the effectiveness and interest in a system such as MINT derives from the structural relationships of the components, rather than from the components themselves. Thus, in this text, we have tried to develop an understanding of how MINT fits together. This may initially seem to impede learning, where compartmentalization is always a strong temptation. However, we believe that the end result will be found to be beneficial. The complete MINT system is more significant than the sum of its parts.

1.3 Purpose

The purpose of MINT is to facilitate the analysis and transformation of structured symbolic information. An example of such analysis and transformation is a conventional language compiler. Other examples include text-editing routines (such as those given in Section 12.6) or interactive dialog systems for specific applications.

In order to satisfy a wide range of possible requirements, the system is organized in the form of a set of general-purpose tools. These tools may be used for many purposes, including the development of new tools. The system itself is constructed by means of the tools which it provides for general use. The open and modifiable structure of the system is essential to its generality, and allows a holistic approach to many problems which previously required ad-hoc solution methods.

Due to its compact structure, MINT is well suited for use in very small machines. An eight bit processor with 32K (here, and throughout, K is used to mean 1024) bytes of storage is sufficient for many purposes. However, the system also operates effectively on large-scale systems. The change made in MINT-3 to use 32 bit storage units permits use of a 32 bit virtual address space, i.e. $2^{32} - 1$ storage units.

The complete machine independence of the MINT language permits the writing of systems which may have wide applicability and permanent value.

1.4 Background

Historically, computing has developed from a primary interest in the algorithms required for solution of numerical problems. The earliest forms of computing were characterized by relatively large algorithmic programs which operated on relatively small quantities of data. As computing technology developed there was a tendency to apply the tools which were developed for this structure to other, often non-numerical, problems. At the same time, the volumes of data, both numerical and symbolic, began to grow very rapidly. At present it is frequently the case that the amounts of data to be processed far exceed the size of the processing programs. Usually, the purely numerical processing accounts for only a very small part of the total. Thus, it is natural to question the basic structure of current computing tools, based as they are on conditions which no longer prevail. It is clear that if programs are used to process very large quantities of data, the value of the program and the importance of correctness of the program are increased. It is also evident that much of the complexity of current computing derives from the attempt to develop algorithms which express symbolic transformations. Finally, the slow and cumbersome operation of early computers provided the incentive for investment in improved efficiency of program execution. Early programs were often operated without substantial change for long periods. Thus, substantial effort in writing and understanding the program could be justified. The speed and flexibility of current computers imply that the limiting factor in their productive use is the rate at which information which is understood by humans can be precisely and correctly communicated to and from the computing system.

These background considerations have led us to attempt to develop new language tools which are based on the view that data transformation is the fundamental task, and that machine independence, readability, effective structure, evident correctness, and precision are essential.

1.4.1 Principles

Current developments in computing suggest that the following principles should form the basis of an effective set of tools.

1.4.2 Data Transformation

The fundamental task in most computing applications is the transformation of the input data into the output data. The means by which this transformation is carried out are of no direct interest to the computer user. This principle implies that the main task in computing should be the declarative task of stating the form of the input and the desired form of the output. If an algorithm is necessary to carry out the defined transformation, this should not be made evident to the user.

1.4.3 Machine Independence

Due to the continuing proliferation of distinct computing systems, it is very probable that a given user task will be desired to be performed on several different computers. The only way in which such operation can be carried out efficiently is to have a completely machine independent means of expressing the intended task. Such machine independence will permit software to attain real economic value, as the software will no longer be restricted to a specific set of hardware.

1.4.4 Readability

The text which causes a computer to carry out a desired transformation must be written, read and understood by people. The present importance of data transformations is such that these computations should not be carried out by programs which are imperfectly understood. In addition, readability must not be in conflict with efficiency or compactness of the resulting machine program. This conflict is evident in many current language systems, but is not present in MINT.

1.4.5 Structural Clarity

The principle of structural clarity is closely related to that of readability. A computing system must induce clear structure in the text which is written for that system. A clearly structured program should be more efficient than a poorly structured one. This implies that it should be easier to express the required task in a well structured manner than otherwise. In particular, there should be no syntactic, or execution efficiency, penalty attached to the use of easily understood procedures to carry out elementary steps in program execution. Of even greater importance, the use of structured data should lead to increased ease of program writing, more readable results, and

higher execution efficiency. The structured data should be easily understood in terms of the user-view of the computing task.

1.4.6 Self-Realization

Any truly effective system must be self-realizing in the sense that it is based on the facilities which it provides. If a facility is provided, there should be only one such facility which is used by the system and by users alike. This principle is important for compactness, but is also essential for correctness.

1.4.7 Precision

The basic components of a language system must be such that they are subject to exact definition and complete verification of correct operation. This, in practice, implies that the basic components must have a high degree of *orthogonality*. Components are orthogonal if they are defined in such a way that there is no dependence between the two definitions. Orthogonality implies that each basic component may be separately verified.

1.5 MINT Functional Structure

Figure 1-1 depicts the general structure of the MINT system, and illustrates the means by which MINT achieves machine independence. The Virtual Machine Interface is precisely and compactly defined. It is independent of any specific Virtual Machine implementation. It is exhaustively tested by the VM Diagnostics. Thus, all information which is above the Virtual Machine Interface line in the Figure is entirely unaware of the underlying host system. It sees only the Virtual Machine Interface. The main task in implementing MINT on a new host system is the creation of a VM(M) Virtual Machine which matches the particular Host System Interface. Due to the compact and orthogonal definition of the Virtual Machine Interface, it is a simple task to create a new verified VM(M) Virtual Machine.

Note also that dashed lines are used to separate the various components of the MINT System. This is intended to indicate that the Language System is entirely flexible in terms of access to facilities and definition of structure. The user may write applications at a low or a high level and may create or exclude facilities as specific application requirements may dictate. This *organic structure* is discussed briefly below, and is a main theme throughout the remainder of this monograph.

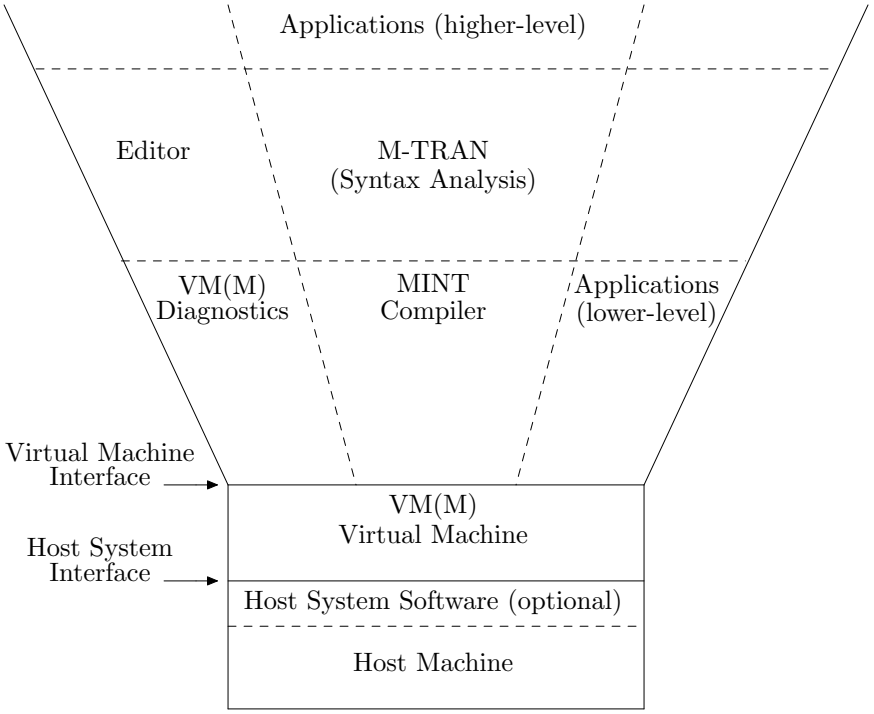


Figure 1-1 MINT Structure

The example extensions of the system shown in Figure 1-1 are M-TRAN, a syntax analysis system suitable for a wide range of text analysis or compiler applications, and an editor which provides a convenient means of creating and modifying text files. M-TRAN is described in Chapter 11 and the editor is described in Section 12.6.

1.6 Organic Programming

The MINT system is structured in such a way that all its facilities may be available to the programmer at all times. No strict distinction is made between *compile-time* and *execution-time*. For this reason it is customary for the compiler to be resident at execution-time with all its functions potentially available as a run-time system. In such an environment all user programs become conceptual extensions to the MINT compiler and may

take control of it if appropriate, or conversely they may act as new language features tailored to a particular application. This is termed an *organic* environment.

While in many cases the organic structure results in a system with increased scope, it is also possible to reduce the set of available facilities. Such reduction is useful in order to provide a well controlled environment for many application systems. Both the extension and the reduction are normally reversible so that the environment may be adjusted to suit current requirements.

1.7 The Dictionary System

The basis for the organic structure of the system is the *dictionary*. The compiler is driven by the input mechanism which attempts to match input sequences with the current set of dictionary entries. If a match is found the compiler carries out the actions associated with the class to which the matched dictionary record belongs. All dictionary items which are to be treated in the same manner by the compiler are declared to belong to the same identifier *class*.

Dictionaries are declared by use of the DICT directive. Dictionary names are maintained in a list and the user may declare which dictionary is to be used for new introductions and searches. The programmer may modify the entries in dictionaries by insertion, renaming, and removal of identifiers. He may likewise define or modify the actions which are associated with each identifier class. He may also define new classes. Thus, the behavior of the system may be made to satisfy a wide range of requirements.

Since the dictionaries control the entire system there is no strict distinction between compiler text and user written text. Since a dictionary may be modified at any time, there is no strict distinction between compiler translation of text and execution of user written text.

1.7.1 Dictionary Entries

The item name contained in a dictionary entry is termed an *identifier*. Identifiers are composed of sequences of character codes. All codes in the ISO/ANSI seven-bit code are allowed including the control codes, such as carriage-return. This generality of identifier construction allows all identified objects to be normal dictionary entries. Thus, all actions are determined by matching of character strings with dictionary identifiers.

1.8 Uniformity

The dictionary system provides one level of uniformity in the system. Another level of uniformity is provided by the fact that many of the functions which the compiler uses as part of the compilation process are in fact general-purpose functions which are also available to the user. Thus, in general, no duplication or specialization is imposed on the application programmer. All of the tools used in construction and operation of the compiler are uniformly available.

1.9 Compactness

All components of the MINT system have been designed to be compact. This has been done in order to achieve a system which can be well understood and which will be efficient in both space and time requirements. A total memory size of 10K 16-bit words is sufficient to operate the entire 16-bit system. It is usually found that less object-text space is occupied, and fewer instructions are executed by procedures written in MINT than the corresponding procedures written in the assembly language of typical general-purpose computer systems.

Compactness is also achieved by means of a low level of redundancy. This implies that MINT source text should be written and read with care. This care will be rewarded by exceptionally clear, concise, and efficient implementations.

1.9.1 MINT System Size

One quantitative indication of the compactness of MINT is the amount of storage used by the system components. These amounts are shown in Table 1-1.

The following points provide interpretation of this Table:

1. Identically the same MINT compiler is used on all machines. Consequently, the size of the compiler does not vary from one installation to another.
2. The size units are currently 32 bit words.
3. Dictionary records can be reclaimed if the facilities to which they give access are no longer required.
4. The size of the VM(M) Virtual Machine will reflect the power of the host machine's instruction set, the size of the system's run-time li-

braries, and the goodness of fit between the host machine and the VM(M) Virtual Machine.

Table 1-1 Store Usage in the MINT System

Component	size in K-words
MINT compiler (machine independent)	7.6
Program space used	4.3
Data space used	11.9
Virtual Machine (system dependent: (gcc-4.1.1 i386 value used)	
With gdb diagnostic data	54K bytes
Without gdb diagnostic data	27K bytes

The complete compiler system source listing is approximately sixty pages long, including the text for auto-compilation. A full compilation listing of the compiler may be produced using the procedures explained in Chapter 14.

1.10 Storage Organization

The storage (or memory) which is made available by the Virtual Machine consists of two blocks of contiguous storage units. The entire addressable space is referred to as *VSTORE*. One block of storage starts with address zero and contains storage units of at least 16 bits. The other storage block was in the past (MINT-2) expected to start at address 32K and contains storage units of at least 8 bits. The first block is referred to as *data-space* and the second block is referred to as *procedure-space*. Previously, each of these blocks could contain up to 32K storage units. Since MINT-3 uses 32 bit addresses, procedure-space can start at an address high enough to accommodate a large data-space and free-space. Starting at the high end of data-space is an area called *free-space*. The allocated area of free-space expands upward (as shown in Figure 1-2) toward the boundary of allocated data-space. While a MINT program is running, it may adjust the data-space and procedure-space pointers to provide allocation in other regions of the total available memory. Figure 1-2 illustrates the layout of these areas and the manner in which space is allocated within each area. Data-space, controlled by the compiler pointer variable DLOC, is used for space allocation of data variables, macros, and character strings. Procedure-space, controlled by the compiler pointer variable PLOC, is used for allocation of the compiled object text resulting from procedures. Free-space is used for buffer areas as explained in Chapter 9, and for dictionary storage. The

values N_d and N_p are the upper limits of data-space and procedure-space respectively. These values are set by the Virtual Machine. They may be determined by the size of problems or the amount of available memory. This is more fully explained in Section 13.2. If the compiler is loaded into VSTORE it occupies approximately the first 4300 words of data-space, the first 7600 words of procedure-space, and about 110 words of free-space. These numbers are as given in the first three entries in Table 1-1. If the Virtual Machine implementation permits use of *virtual memory,* a large space may be acquired for the MINT Virtual Storage. In this case, N_d and N_p may be given fairly large values, and a very large area above N_p may be made available for application use. Specifically, the reference Virtual Machine, described in Ch. 15, is configured to provide a total of 16M words of Virtual Storage. The storage required for the Virtual Machine interpreter is not a part of Virtual Storage.

1.11 The Virtual Machine

The basis for the execution of all MINT text is a *Virtual Machine*. A Virtual Machine is defined as a composite of hardware and software which presents an execution environment which satisfies a specified (Virtual Machine) definition. The Virtual Machine definition required for MINT operation is given in detail in Chapter 13. The basic structure of the Virtual Machine is given in Section 4.2. Since the MINT system is entirely written in MINT, the single Virtual Machine operates the compiler system and all user written text. The Virtual Machine which implements this environment is referred to as the VM(M) Virtual Machine.

The *object* text which is created by the compiler, and executed by the Virtual Machine, is defined in terms of instruction words, integers, and character strings. The object text also has an ISO/ANSI character representation for external storage and for transportation between systems. This format is termed the *portable format*.

The portability of the MINT system is achieved by definition of a Virtual Machine with a compact instruction set which is easily mapped onto most real machines. The Virtual Machine is what is termed a stack-based reverse-Polish machine. These terms will be explained in subsequent Chapters.

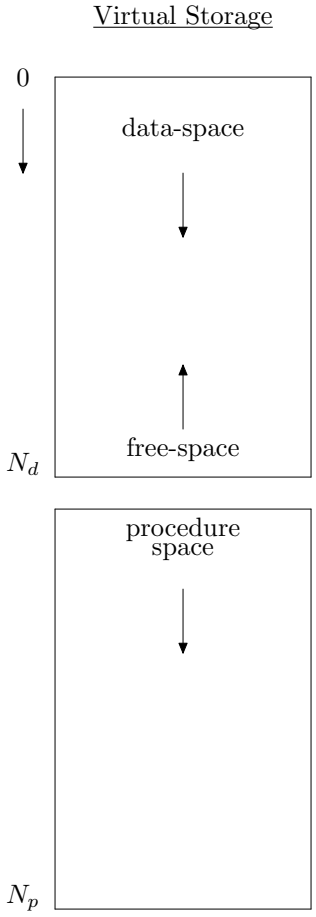


Figure 1-2 System Storage Allocation

The Virtual Machine may be implemented on a *host* (real) machine in several ways; by interpretation of the Virtual Machine object text, an approach which is very fast to implement; by means of an object text generator appended to the MINT compiler which generates object instructions for the host computer; or by micro-coding the VM(M) Virtual Machine instruction set. Virtual Machine interpretation has become by far the most frequently used implementation method. This approach minimizes the amount of non-MINT language or special-purpose text and produces, in most environments, a very efficient system. If the host system is already accessible

(i.e. it has a file system, an editor, and a suitable language processor) the VM(M) Virtual Machine interpreter can be implemented directly on the host machine using whatever language is most suitable. Then, the portable format loader may be implemented in a similar way. This completes the implementation. If the host system has no (or inadequate) software facilities, another host system should be used. In this case it is usual to write a basic assembler in MINT for the new host machine and write the Virtual Machine interpreter in this assembly language. The output of the assembler may be portable-format host instructions. (Portable format is described in Section 13.3 and 13.4.) Then the only remaining task is to write the text loader for the host machine so that this machine can load data in portable format. A complete implementation, using any of these approaches, should not require more than a few weeks' work.

Chapter 15 describes the reference Virtual Machine implementation in C. The source code for this implementation is included in the MINT distribution. The source code should compile without problems on most contemporary systems, such as Linux.

1.12 Introductory Examples

Some of the characteristics and power of MINT can be appreciated only after study and experience. However, simple uses are easy to learn. The following examples should be understandable even at this point.

In MINT, a procedure which is to be obeyed when the compiler encounters its name is called a directive (DIR). In fact, much of the compiler itself is composed of directives. For example, we might write the following directive, whose name is EVAL:

```
DIR EVAL: ENTRY,
          IPAR,
          OPINT( ),
          OPNL, EXIT .
```

When this directive is referenced the first action is to reference the procedure IPAR. IPAR is a compiler procedure which reads the next expression from the input source, evaluates the expression, and leaves the result on the operand stack. (The operand stack is a storage area used for instruction operands and for procedure parameters. This is more fully explained in Section 4.2.) The procedure OPINT simply prints the value found on the operand stack. OPNL sends a carriage-return character to the output stream. This closes the current line of print. EXIT causes a return to the point at which the current procedure was referenced. Thus, if we have

defined and set two variables by:

```
VAR X: 25
VAR Y: 19,
```

we could then write:

```
EVAL(X+Y)
```

and obtain the result

```
00044
```

printed as output. The details of how and why this example works will be fully explained in subsequent Chapters. The example itself is used again in Chapter 12 where it is more fully explained.

A similar result to the above example may be obtained by a seemingly quite different construction:

```
NOW OPINT(25+19), OPNL ! .
```

This line is treated in the following way by the MINT system. When the NOW directive is encountered the compiler treats all text up to the next ! (also a directive) as text which is to be immediately compiled, executed, and then discarded. Thus, the OPINT(25+19), OPNL sequence is compiled and executed. This causes the result 00044 to be printed, exactly as in the previous example. After execution, the space used by the generated object text, and all record of the compilation, is discarded.

Yet another example construction is:

```
MACRO PRINT: 'NOW OPINT(X), OPNL !' .
```

This text simply causes the compiler to save the string

```
NOW OPINT(X), OPNL !
```

as a MACRO whose name is PRINT. After this definition and setting of PRINT, if the identifier PRINT is provided as input, the compiler will substitute the string

```
NOW OPINT(X), OPNL ! .
```

This will result in immediate compilation and execution just as if the string had been entered directly. Thus, the value of the variable X will be printed.

At this point an aspect of MINT generality should be evident. Conventional procedures may be written and executed after the compilation

process is complete (these are called MINT functions). Alternatively, directives may be written which are executed immediately upon occurrence of a reference to the directive, or immediate compilation and execution may be obtained by use of the NOW...! construction. Finally, compilation may be deferred until later by means of MACROs. No restrictions are placed on the use of these facilities. They may be used in whatever context or combination may seem effective for a given application.

Note that in some cases, such as those above, the standard function and argument notation such as

```
OPINT(X+Y)
```

is used. However, since the operand stack is always used for procedure parameters, the above text could equivalently be written as

```
X+Y, OPINT .
```

The + operator leaves its result on the stack, so that this result is available as the parameter of the OPINT reference. The compiler, in effect, carries out just such a rearrangement on the standard notation. It is good practice to indicate that a procedure expects arguments by means of parentheses and commas even if some of the parameters have been obtained previously. Thus, the above text would normally be written as

```
X+Y, OPINT( ) .
```

The use of the stack and the role of parentheses and commas will be more fully explained in Chapter 4.

2. MINT Language Components

2.1 Introduction

This Chapter provides a general introduction to the MINT language and the facilities available in the MINT compiler. Subsequent Chapters give more detailed and precise definitions of each component of the system.

2.2 Definitions

The major constructs of the MINT system which require definition at this point are the Virtual Machine primitives, the IPAR mechanism, and the object classes provided by the compiler.

2.2.1 Primitives

The Virtual Machine *primitives* are the instructions which are carried out by the VM(M) Virtual Machine. These primitives are fully described in subsequent Chapters. Chapter 13 gives the precise definition of each primitive in a form suitable for the implementation of the Virtual Machine. The term *operator* will be used to refer to primitives in many cases. However, operators include directives, functions, and macros as well as primitives. No special syntax is used to distinguish between operators. For this reason an operator may be a primitive in one MINT implementation and a function in another implementation without causing any incompatibility.

2.2.2 The IPAR Mechanism

The IPAR (Input PARameter) mechanism is used to compile and evaluate input sequences. This is the facility used by the compiler to obtain evaluated results. For example, the syntax action which results from the recognition of a known identifier may cause the compiler to reference the IPAR mechanism. This in turn will cause the compiler to compile and exe-

cute the subsequent expression from the current input stream, and return any resulting values on the operand stack. When the IPAR procedure is referenced, the next input sequence is acted upon as if it were enclosed by a NOW. . .! sequence. The user may make use of the IPAR mechanism whenever there is a need to obtain the result of compilation and execution of a sequence of characters from the input stream.

2.2.3 The Class Construct

The *class* construct permits the definition of the fundamental structures which determine the actions of the MINT compiler. It also permits the declaration by the programmer of any syntactic constructs which he may choose. Normally, the compiler is driven by its input mechanism which attempts to match input sequences with the current dictionary entries. When a match is found, the compiler carries out the syntax action as determined by the class of the matched item. When no match is found to exist in the dictionary for an input sequence the compiler normally rejects the input with an appropriate diagnostic. However, a mechanism exists which causes input sequences to be converted to identifiers and inserted in the dictionary. The use of this mechanism is referred to as the *introduction* of an identifier. An identifier must always be introduced into a specified class. The syntax action which is referenced when the identifier CLASS is matched is a procedure which introduces the name of a new class of identifiers. The directory record which is created by the class syntax action is a record which contains the address of a record which contains the attributes of the newly introduced class. The structure of the class construct allows the source text for the MINT compiler to introduce the class construct by means of the construction: CLASS CLASS.

2.2.3.1 Class Attributes

The introduction of an identifier into the dictionary causes the attributes of the appropriate class to be set in the new dictionary record. Although the number of attributes is variable, the MINT compiler normally associates three attributes with each identifier. Together, these three attributes enable a wide range of syntactic structures to be analyzed, and translated into text which can be executed by the Virtual Machine, or which may be further transformed for other forms of execution. The attributes are briefly described below:

Syntax Action: The *syntax* action attribute determines the action to be performed by the compiler when the identifier is recognized in the input stream.

Assignment Action: After an identifier has been introduced, data may be associated with it. The data may take the form of an arithmetic value, a character string, the address of an object-text sequence (procedure), or a specified address in virtual storage. This association is referred to as *assignment*. The assignment action is determined by means of the assignment action attribute.

Generative Action: The *generative* action attribute determines the action to be taken when the compiler is called upon to generate object instructions or data as a result of the compilation of a reference to an identifier. Not all identifiers require a generative action attribute since all required actions may have been performed by the syntax or assignment actions.

In addition a *precedence* attribute is associated with each identifier. The precedence attribute is a number which determines the sequence in which the generative actions are performed. The precedence number is recorded in the identifier's dictionary record. The default precedence is determined by the class of the identifier, but the value may be overridden by use of the PRIORITY directive (See Section 4.9.2.). The default precedence for all classes shown in Table 2-1 is zero, except for the CLASS PRIM. The default precedences for PRIM re given in Table 4-1. The use of precedence is explained in Section 4.6.

2.2.4 Initially Defined Classes

In addition to the class CLASS itself, the following classes are defined in the compiler and are available for general use. These classes are used within the compiler and are sufficient for many purposes, such as compiler writing. However, new classes may be introduced as needed. The compiler defined classes are:

2.2.4.1 DICT – Dictionary

The MINT dictionaries provide the information which causes operation of the entire system. In order to provide control over the currently known sets of identifiers the identifier records are composed into distinct dictionaries. These dictionaries are composed into a list. The list of dictionaries may be manipulated by referencing dictionary names and by procedures which are described below. New dictionaries are created by the DICT construct. The operators \ and % may be used to reference a named dictionary. The dictionary list may also be referenced by use of the MINT list operators. Pointers into the list structure control the way in which the various dictionary manipulations are carried out. Thus, it is possible to select which

dictionaries are searched and which dictionary is used for introduction of new identifiers.

2.2.4.2 PRIM – Primitive

A set of operators (or primitives) is defined in the class *primitive*. These are the Virtual Machine instructions. Each identifier in this class causes the compiler to generate a single Virtual Machine instruction. Each Virtual Machine instruction is represented by a unique number between 1 and 80. Chapter 6 introduces the basic primitives, while Chapter 13 gives a full description of the instruction codes, formats, and operation of each primitive. Section 13.6 contains a Table of all compiler defined primitives. It is straightforward to introduce a new primitive at any time. However, the Virtual Machine must be extended so that it will correctly execute the new primitive instruction. For this reason primitives are more static than other classes.

2.2.4.3 DIR – Directive

A *directive* is a procedure which is performed when the compiler encounters a reference to its identifier. Parameters may be passed to a directive by means of the IPAR mechanism, as described in Section 8.2. The set of directives defined initially within the compiler determines the compiler's standard actions. These initial directives may be manipulated in exactly the same manner as directives defined within user text.

2.2.4.4 FN – Function

A *function* is a procedure which is performed when a reference to its identifier is encountered by the Virtual Machine. Functions may have parameters which are passed by means of the operand stack. A referenced function normally returns control to the point of reference by means of the EXIT primitive.

Functions and directives are both procedures which are defined and written in exactly the same manner. However, a reference to a function results in compilation of a reference to the function, whereas a reference to a directive causes the compiler to execute the directive immediately. A mechanism also exists which defers execution of a directive so that the directive is treated as a function.

2.2.4.5 MACRO – Macro

The identifier class *macro* provides the means of identifying strings which are to be used as compiler source input when a reference to the macro name is encountered. At the end of macro string processing the compiler resumes processing of the previous input stream. Due to the structure of MINT, no special indicators are used to indicate macro references or arguments.

2.2.4.6 ICON – Constant

A *constant* is a fixed numeric quantity. It may be named or it may be a literal (i.e. anonymous) value. Literal constants are used in contexts where the object need not be referred to by a name. Constant values are composed of a sign and a 31-bit integer.

2.2.4.7 VAR – Variable

The identifier class *variable* is used for single word objects such as arithmetic variables, counters, or address values. Variables, like constants, are treated as being composed of a sign and a 31-bit integer.

The means of implementing this arithmetic definition may vary somewhat depending on the host system. Programming which depends on overflow effects or on the characteristic features of one's- or two's-complement arithmetic should be avoided.

2.2.4.8 LAB – Label

The identifier class *label* is used to name storage addresses. When a label is set its address value is set to the value of the currently active location counter (DLOC or PLOC).

2.2.5 Summary of Class Characteristics

The following Table gives the action carried out by the compiler for each of the contexts in which an identifier can occur, and for each of the compiler defined classes.

Table 2-1 Class Characteristics

Class	Action		
	Syntax	Assignment	Generative
CLASS	identifier introduction	set DATA	GET(label)
DICT	procedure references	set DATA	procedure reference
PRIM	shunt	none	operation code
DIR	procedure reference	set PROG	none
FN	shunt	set PROG	procedure reference
MACRO	switch input	set DATA	GET(macro)
ICON	shunt	none	GET(constant)
VAR	shunt	set DATA	GET(variable)
LAB	shunt	none	GET(label)

The *shunt* action consists of saving a record of the identified object. These records are saved on a *shunt stack* so that they may be processed by the generative procedure according to the precedence rules (as discussed in Section 4.9).

The compiler operates in one of two states, *program* state or *data* state (See Section 4.7.1). The set PROG and set DATA assignment actions cause the compiler to perform the following:

1. The compiler state is set to generate subsequent text into procedure-space (program state) or data-space (data state) respectively.
2. The current location counter value of the appropriate space is associated with the identified dictionary record as its address value. The generative action causes the data value associated with the dictionary record to be generated into the next word of storage as controlled by the currently active compiler state. If the active state is program, a GET or GETV instruction (See Section 6.2.1) will be generated.

2.3 Identifiers

An identifier consists of any sequence of characters from the ANSI character set including blanks and other special or control characters. An identifier names a constant, variable, label, macro, function, directive, primitive, or other class object. Thus,

```
a
A
aa
aA
ab
abc
x
123-
$x 69%
::=
```

are all valid identifiers.

2.3.1 Identifier Matching

When the compiler scans characters from the input stream it accumulates the characters, attempting to match the resulting string with a dictionary entry. This matching is attempted after each character is read. If no match occurs by the point at which the end-of-name (See Section 2.3.3.1) condition occurs, the unmatched identifier error condition is indicated and reading of input restarts with the next line of source text. Otherwise, if a no match occurs another character is read and the match attempt is repeated. If a match occurs, the next character is read and a match attempt is made using this longer string. If this match succeeds another character is read and the match attempt is repeated. If this match fails the previously matched string is accepted as an identifier. Thus, the longest matching string is always found. If a, aa, and aaa have each been introduced each will be recognized.

If compiler defined identifiers are embedded in other identifiers then some constructions may not produce the superficially apparent results. For example,

```
VAR ab      . introduce the variable ab
VAR ab+     . introduce the variable ab+
```

will have the effect that an expression such as

ab+c

will be interpreted as a reference to ab+ followed by a reference to c. The + operator will not be recognized. It is also possible that an unintended construction will cause the compiler to recognize a directive and, therefore, perform some unexpected action. The effects of such action may not be apparent immediately. It is easy to see that the generality of identifier construction is a powerful tool, but one that must be used with care.

2.3.2 Identifier Manipulation Directives

Identifiers may be created by introduction into the selected class, removed by means of the FORGET directive, or renamed by means of the RENAME directive. The syntax for directives which create or remove identifiers is:

directive-name identifier-name

where directive-name is the name of the required directive and identifier-name is the identifier which is to be acted upon. The syntax for the directive which renames identifiers is:

RENAME old-identifier new-identifier

where RENAME is the renaming directive, old-identifier is the previous identifier name and new-identifier is the name which is to be created as a replacement.

2.3.3 Introducing Identifiers

An identifier must be introduced before it can be referenced in any way. An identifier does not exist within the system until it is introduced. There are no default introductions. An identifier is introduced to belong to a defined class. The identifiers for the standard compiler defined classes (as described above) are:

CLASS	- class
DICT	- dictionary
DIR	- directive
FN	- function
ICON	- integer constant
LAB	- label
MACRO	- macro
VAR	- variable.

Thus, the identifier abc may be introduced as a variable by writing

```
VAR abc .
```

Similarly,

```
FN function
```

introduces the identifier function as a function, and

```
LAB begin
```

introduces the identifier begin as a label.

If an identifier is introduced more than once at a given block level, (For definition of blocks, see Section 2.3.5) any reference to it is to the most recently introduced copy. Thus,

```
FN      xyz
FN      xyz
LAB     xyz
VAR     xyz
```

has the effect of introducing four identifiers xyz, with any subsequent reference being to the variable xyz, unless xyz is removed or renamed (See Sections 2.3.3.2 and 2.3.3.3 below.).

2.3.3.1 Identifier Naming

When an identifier is introduced its name is considered to begin with the first non-blank character after the class name, and to end when the following condition is met:

A blank, colon(:) or carriage-return (C/R) is encountered. A special input routine is used by the directives which introduce or manipulate identifier names so that the subject name is read according to the above rule without any attempted matching in the current dictionary. Thus,

```
VAR             abc def
```

introduces variable abc and references the identifier def. If it is required to introduce an identifier with an embedded identifier terminator, the introduction escape character (;) must be used. ;S is replaced by a space and ;CR by a carriage-return (ASNI character code 13). Thus,

```
VAR             abc;Sdef
```

introduces the identifier:

```
abc def .
```

The single blank between c and d is significant and is part of the identifier. The identifier abcdef does not exist unless separately introduced. The characters ":" and ";" may be included in an identifier by applying the escape character to them. Thus ;: and ;; cause inclusion of : and ; respectively.

2.3.3.2 Removing Identifiers

Identifiers may be removed from the dictionary by means of the FORGET directive. Thus,

```
FORGET abc
```

removes the identifier abc. This facility enables the creation of local identifiers:

```
VAR xyz                  . introduce variable xyz
    :
    text using xyz
    :
  VAR xyz                . introduce new variable xyz
      :
      text using new xyz
      :
  FORGET xyz             . remove new xyz
    :
    text using xyz
    :
```

In the above example the first section of text references the introduced variable xyz. After the second introduction all references to xyz refer to the new variable xyz. After the FORGET any subsequent references are to the previously introduced xyz.

Reintroduction of an identifier may be performed to any level, and it is not necessary that the identifier be introduced in the same class each time. When an identifier is forgotten only the name is deleted. Program text or data associated with the identifier are not deleted.

2.3.3.3 Renaming Identifiers

An identifier may be renamed by means of the RENAME directive. Thus,

```
RENAME x$y 1def9
```

changes the name of the identifier x$y to 1def9. The rules for the new name are exactly the same as for introducing identifiers. All attributes (class, address, etc.) of the new name remain unchanged from the original identifier. RENAME is a replacement operation. Thus, the original identifier name no longer exists in the dictionary.

2.3.4 Setting Identifiers

After an identifier has been introduced into the dictionary its dictionary record exists, but is not complete. At some stage the identifier must be assigned a location, or an address value, in the object program. This assignment process is also referred to as *setting* an identifier. Directives and macros may only be referenced after having been set.

2.3.4.1 The Colon(:) Directive

An identifier is assigned an address value by a reference to the identifier followed by a colon(:). Thus,

```
abc:
1$De:
func:
```

cause each of the identifiers which precede the : to be set. Each of these identifiers must previously have been introduced.

If the identifier is a function (FN) or a directive (DIR) it is assigned the current location in procedure-space. If the identifier is a variable (VAR) or macro (MACRO) it is assigned a location in data-space. When a label (LAB) is set it is assigned a location in whichever area the compiler is currently operating. This point will be further clarified in the Section on compiler states (Section 4.7). An identifier defined to be an integer constant (ICON) is introduced and set without use of the colon directive as will be explained in Section 2.5.1.

Notice that the use of : is not entirely consistent with the usual ordering of objects in MINT. The : acts on the object which precedes it, not only on following objects.

2.3.4.2 The EQV Directive

As noted in the previous Section the colon (:) directive assigns the current

value of one of the compiler's location counters as the address value of an identifier. The EQV directive allows any arbitrary value to be assigned as an identifier address value. The form of an EQV reference is:

```
identifier EQV IPAR-expression .
```

Thus, for

```
x EQV 6
```

the address value assigned to x is 6, and for

```
abc EQV (def+10)
```

the address value assigned to abc is the result of evaluating def+10. The construction,

```
x EQV DLOC
```

is equivalent to

```
x:
```

if x is a variable or macro.

2.3.4.3 Immediate Setting

The processes of introducing and setting an identifier may be combined, as in:

```
VAR     x:
FN      abcd:
LAB     19x EQV 7 .
```

If this procedure is not used then the identifier must be set by following a reference to it with either colon or EQV. For example,

```
VAR x

        :
        other text
        :
x: .
```

Note that

```
VAR x:
```

is equivalent to

```
VAR x x: .
```

2.3.4.4 The REF Directive

The REF directive is used to prevent the compiler from carrying out its normal action for identifiers in class DIR or MACRO. A directive name preceded by REF is treated like a function name. A macro name preceded by REF is treated like a variable name. One use of REF is to allow flexibility in setting of identifiers in class DIR and MACRO. Unless directives and macros are introduced and set immediately, they must be set by use of the REF directive. The choices are either

```
DIR      x:   ENTRY
                :
              directive body
                :
              EXIT
MACRO    z:   ' body of macro' ,
```

or

```
DIR      y
MACRO    z
                :
              other text
                :
REF      y:   ENTRY
                :
              directive body
                :
              EXIT
REF      z:   ' body of macro' .
```

REF is also commonly used when it is desired to compile a reference to a directive so that the directive is executed as part of the execution of the compiled text.

2.3.5 Local Identifier Blocks

A local identifier block is a section of text preceded by the BLOCK

directive and followed by the ENDBLOCK directive. Any identifiers introduced within the block are automatically forgotten at the end of the block. For example, consider

```
BLOCK
VAR       a
VAR       b
VAR       c
VAR       d
FN        x
FN        y
      :
      text
      :
ENDBLOCK .
```

When the ENDBLOCK is encountered a FORGET operation is performed for each of the identifiers introduced since the most recent BLOCK directive. Local identifier blocks may be nested to any level, up to the implementation limit of 100.

2.3.5.1 The SAVBLOCK and SETBLOCK Functions.

The BLOCK and ENDBLOCK directives provide hierarchical scope of identifiers. The SAVBLOCK and SETBLOCK functions may be used to remove and reintroduce sets of identifiers at any time. The SAVBLOCK function operates analogously to the ENDBLOCK directive, but instead of permanently forgetting the identifiers at the given block level, it links them into a list whose starting location was provided as the parameter. As with ENDBLOCK, the block level is decremented by 1. Thus, consider

```
VAR       ss1:0
DIR       s1:   ENTRY
                SAVBLOCK(@ss1), EXIT
BLOCK
FN a
VAR       b
                :
                s1 .
```

(The construction @ss1 generates an address constant as defined in Section 2.5.2.) When the s1 directive is referenced, the identifiers introduced after the BLOCK directive (a and b) are removed from the dictionary and linked

to location ss1.

The SETBLOCK function operates analogously to the BLOCK directive in that it increments the block level to be used in all subsequent identifier introductions. In addition, SETBLOCK will take all identifiers chained to the supplied variable address and reintroduce them into the dictionary at the new level. Thus,

```
DIR     r1:  ENTRY,
             SETBLOCK(@ss1), EXIT
             r1
```

will increment the block level and reintroduce the variables a and b.

By using SETBLOCK and SAVBLOCK functions, the set of known identifiers can be freely manipulated. In particular, sets of identifiers may be created, and made known within various blocks at arbitrary block levels.

2.4 Internal Compiler Identifiers

Under normal conditions only the identifiers which are described in this monograph are accessible to programs. However, the compiler contains many internal procedures which are useful for more advanced programming (such as compiling the compiler). The UNLOCK INTDIC directive may be used to make these internal identifiers available. The LOCK INTDIC directive is used to remove them. The ICL$ and RCL$ directives were used in previous versions for UNLOCK INTDIC and LOCK INTDIC respectively.

2.5 Constants

A constant is a sequence of digits or characters which have a fixed (constant) value. Constants may be literal (i.e. anonymous) or they may be associated with an identifier. There are several types of constants as described below.

2.5.1 Integer Constants

An integer constant is composed of a string of integer digits. For example,

```
14628
3
18
```

are integer constants, unless otherwise identified. The magnitude of an integer constant must not exceed $2^{31} - 1$. A negative integer constant is produced by preceding a string of digits with the directive MINUS. Thus,

```
MINUS 24
```

yields the constant value −24. The MINUS directive thus acts as a unary operator which transforms an integer constant to its negative value. The MINUS directive must be distinguished from the NEG and - operators which perform arithmetic on variables. These operators are discussed in Section 4.8.1.

If a string of digits has been introduced as an identifier then the occurrence of that string of digits will be interpreted as a reference to the identifier and not as an integer constant. Thus, after the introduction

```
VAR 1234,
```

```
1324        is an integer constant,
123         is an integer constant,
1234        is a variable reference.
```

Identifiers may also be defined as integer constants. The form of such definition is:

```
ICON identifier IPAR-expression.
```

Thus, given

```
ICON    x    5
ICON    y    MINUS 3
```

when the identifier x or y is subsequently referenced, it will be interpreted as an integer constant.

Note that the constructs

```
ICON    x    5
```

and

```
LAB     x    EQV 5
```

yield the same results in most contexts. However, the second construct is not entirely appropriate for an integer constant since identifiers in the

class LAB are treated as address constants. If object text is moved from one VSTORE address to another, all relevant address constants must be adjusted to the new address base. Such adjustment would not be applied to identifiers in the class ICON.

2.5.2 Address Constants (@)

An address constant is formed by preceding an identifier with the directive @ (at-sign). Thus,

```
@x
```

forms a constant whose value is the address of the identifier x. The at-sign is a compiler directive which causes reading of the following identifier and obtains the address value of the identifier. Thus, the constructs:

```
VAR xx:2                    . line 1
LAB xa EQV @xx              . line 2
LAB xv EQV xx               . line 3
```

will have the following results. Line 1 will introduce and set the variable xx, and set 2 as its data value. Line 2 will introduce the label xa and set its address value to the address value of xx. Line 3 will introduce the label xv and set its address value to 2.

Note that since the @ directive reads the following identifier no normal compiler action is taken as a result. Thus, for instance, if the identifier following the @ directive were a name of a macro, no macro substitution would take place. Instead, @ would obtain the address value of the identifier which would be the address of the macro body. Also note that these address values are treated as unsigned integers on the range 0 to $2^{32} - 1$. Only address arithmetic operators (ADIFF and FROM, see Section 4.8.3) may be used on these quantities.

2.5.3 Character Constants (#)

A character constant is a literal whose value is the integer value of a single character. It is formed by immediately preceding the character with the directive # (hash) without any intervening blanks. Thus,

```
#x
```

yields the integer value of the character x, which is 120. In the expression

```
#0+6
```

the integer value of the character 0 is added to the integer constant 6. This results in the value 54, since the ISO character code value of 0 is 48.

2.5.4 Evaluated Constants (&)

An evaluated constant is the result of an expression which is evaluated at compile time. The directive & causes evaluation of the following IPAR-expression. Thus,

```
&(8 FROM @table)
```

results in an evaluated constant whose value is the address of the eighth item after the start of table. (The FROM operator computes address offsets as explained in Section 4.8.3.) This is useful for implementing the practice known as *parametric programming.* The following example illustrates this usage:

```
VAR     table:
        :
        table entries
        :
LAB     tabend:
```

The identifier table is set to the address of the beginning of the table and tabend is set to the address immediately following it. A constant whose value is the length of the table can be formed by the construction:

```
&(@tabend ADIFF @table)
```

where ADIFF is the operator which computes the difference between the two addresses. All identifiers which are used in the construction of an evaluated constant must have been both introduced and set prior to such use.

2.5.5 String Constants (' ')

A string constant is a string (strictly, a string address) which may be referenced as a literal. Such a constant would typically be used as a parameter to a function which operates on strings (See Chapter 10). For example the compiler's string output function OUTST may be referenced by:

```
OUTST('string')
```

where the characters between the single quotes form a string constant. The single quote character (')[1] may be included in a string by preceding it by the escape character, ; (See Section 2.3.3.1 and 10.5.9.). Similarly, the escape mechanism may be used to include the carriage-return and form-feed characters in string constants. When an escape character is encountered in a string and the immediately subsequent character has no defined escape meaning, the character following the escape is accepted as input. Thus the sequence ;; may be used to include a semicolon in a string.

Strings may continue over any number of lines. In order to provide line indentation for readability the following rules apply to continuation of multi-image strings:

1. A line logically terminates after the last non-blank character.
2. The second and succeeding lines logically start following the first non-blank character.
3. The string is terminated by a closing quote in the normal manner.

Thus,

```
'This string
        ' is continued on
        ' several input
        ' lines.' .
```

results in the string:

This string is continued on several input lines.

2.6 Diagnostics

Due to the flexibility of expressions in MINT there are few diagnostics which result from improperly formed expressions. In general, input which is not in the form which was intended results in recognition of an undefined identifier. The only specific syntactic error is the use of a closing parenthesis which is not matched by an opening parenthesis. The complete set of diagnostics is described below.

2.6.1 Undefined Reference

If at any time the identifier end condition occurs (See Section 2.3.3.1.) and

[1] The intended quote symbol is usually shown on a keyboard as the right single quote ' shown as a vertical quote, but some keyboards may show the symbol differently.

no match has been found in the dictionary and the characters do not form an integer constant, then a diagnostic will be printed. After this condition is recognized the compiler executes the . directive so that any remainder of the current line is discarded. Normal compilation continues at the next line.

2.6.2 Identifier Reset

If the compiler encounters the identifier setting action for an identifier which has already been set it prints a warning diagnostic to indicate this fact. The identifier is reset, i.e. the setting action is performed.

2.6.3 Unmatched)

The compiler records the level of nesting of parentheses. Each opening parenthesis increments the level and each closing parenthesis decrements the level. If a closing parenthesis is encountered when the level is zero a diagnostic is printed and the closing parenthesis is otherwise ignored.

2.6.4 SAVBLOCK/ENDBLOCK with no Active Block

The SAVBLOCK and ENDBLOCK functions can only be used within the scope of a BLOCK. If the current block level is zero a diagnostic is printed and the function is EXITed without any other action.

2.6.5 Storage Overflow

Available storage may be exhausted when data-space or procedure-space are no longer available, or when the data-space and free-space areas overlap. For both conditions, corrective action requires a knowledge of the amount and structure of storage for the particular implementation that is being used. Chapter 14 gives the details of storage configuration.

2.6.5.1 End of Store

If a condition arises such that the compiler is requested to store data into a location whose address exceeds the highest configured data storage address, a diagnostic is printed and an ESTOP instruction is executed. Likewise, if the compiler is requested to generate object text into procedure-space which exceeds the upper limit of procedure-space (N_p), a diagnostic is

printed and an ESTOP instruction is executed.

2.6.6 End of Free-space

If a condition arises such that a NEXTFREE function (See Section 9.4.1) cannot be satisfied due to lack of available memory, a diagnostic is printed and an ESTOP instruction is executed.

2.7 Problems

2.7.1 Problem 1

Define identifier introduction and identifier setting. State reasons why introduction and setting are separated. Give specific examples for variables, functions, labels, and directives.

2.7.2 Problem 2

Write the MINT text to introduce a variable abc, a label LABEL$, and a directive CR-LF.

2.7.3 Problem 3

If a variable identifier abc has been introduced, explain how LAB abc will be treated. Why is abc not recognized as a variable identifier in this case?

2.7.4 Problem 4

What characteristics distinguish the use of literal constants from the use of identified objects?

2.7.5 Problem 5

After the sequence VAR 15:25, FORGET 15 how would a reference to 15 be treated? What happened to the data allocation for the value 25?

2.7.6 Problem 6

Construct a data table structure which contains 5 entries with each entry made up of 3 items using parametric programming so that the number and length of the entries may be varied.

2.7.7 Problem 7

Convert the text of this problem into a MINT string whose name is prob2-7. Use more than one line to express the string.

2.7.8 Problem 8

Write text which will cause the undefined reference diagnostic to be printed.

3. Program Listing Control

3.1 Introduction

MINT source text is free-format. Thus, the programmer may arrange the source language in a manner that is most convenient in terms of input and listing facilities, updating, and readability. It is standard practice to decide on a layout for the source language that tends to display the logical flow of the program in a consistent and readable manner. Indentation, pagination, and spaces between meaningful sections of text are often used for these purposes.

To aid in the orderly presentation of compilation output, directives are provided which control choices of listing output, page formatting, and page titles.

3.2 Listing Options

Four listing control directives are available:

LIST
LOCS
LCODE
NOLIST

The directive LIST causes each input image to be displayed during compilation, prefixed by its line number. The line number is displayed as two fields, separated by a period. The first field is the current value of SIUNIT (See Section 10.5.1). The second field is the line number within the file associated with the current SIUNIT (i.e. the file named on the current SI directive). The directive LOCS (which usually follows the LIST directive) causes the display line to be expanded to include the location counters along with the line number and text. The procedure-space location counter is displayed first followed by the data-space location counter. The counters are only displayed if the value has changed from the previously displayed value. Since the floating process, described in Section 4.6, can apply across input images, the values of these location counters cannot be taken to be exact. The displayed value is the value which is current when the image is

read. However, at this point text from previous lines may not have been generated. Thus, the displayed values may be somewhat "behind" the location of the text. The LOCS directive has no effect if the LIST option has not been selected. The LCODE directive causes display of the generated VM(M) object text which results from each input line. Again, due to the floating process, an identifier in one line may appear in the generated object text for a subsequent line. In order to ensure correct sequencing of the compilation output, the directive LOCS must precede LCODE if both are used. Thus,

```
LIST LOCS LCODE
```

will cause listing of the input images, the data and program location counters, and the generated VM(M) object text. The NOLIST directive disables all listing options.

3.3 Comments and Pagination

If the period (.) directive is referenced in an image, its effect is to cause the compiler to disregard all following text in that line image. The full line image is displayed if the LIST directive has been previously referenced. The period symbol may, of course, be used freely in any context such that it is not recognized as an identifier.

The PAGE directive causes the compiler to skip to a new page in the output listing if any listing options are in effect, and to display any TITLE information at the top of the new page. The directive itself is displayed on the current page.

3.4 The TITLE Directive

The TITLE directive permits source listings to be formatted as titled pages. Each page consists of a header image and a variable number of lines. The header image consists of a text string, a page number, the current date, and the current compiler level. The TITLE directive requires three parameters: the number of lines to be displayed per page, the initial page number, and the address of the header text string. Thus, the sequence:

```
VAR x: 'Header message.'
TITLE (52,1,@x)
```

will cause the string "Header message." to be displayed at the top of each page. Pages will be sequentially numbered starting with number 1, and

each page will contain up to 52 lines of text.

If the specified page number is zero, the existing page number will remain in effect. If the first parameter is specified to be zero, the display of page headings is discontinued, and the NOLIST directive is referenced. The TITLE directive automatically references the LIST, then PAGE directives. It does not reference LOCS nor LCODE.

3.5 Problems

3.5.1 Problem 1

Write the text which would initiate titled, page-formatted listing, skip two pages, and discontinue the listing.

4. MINT System Structure

4.1 Introduction

This Chapter describes the organization of the MINT system, basic properties of the compiler, and the definition and manipulation of MINT language expressions.

4.2 Basics of the VM(M) Virtual Machine

This Section gives a very brief introduction to the VM(M) Virtual Machine. This introduction is sufficient for understanding of basic MINT programming. The full definition of the Virtual Machine is given in Chapter 13.

4.2.1 VM(M) Stacks

A *stack* is an area of storage which functions as a push-down list. The *stack pointer* is the address of the top item on the stack. The stack must always be addressed through the stack pointer. When an item is added to the stack the previous item is pushed down and the new item becomes the top one on the stack. If the top item is removed, the previous item becomes the top item. The verb *obtain* is used to refer to *pushing*, or adding, an item on the stack. An item which is removed from the stack may be referred to as having been *popped* from the stack. The items which are manipulated on the stack are referred to as *objects*.

The VM(M) Virtual Machine contains two stacks, one for procedure linkage and one for *operands*. Operands are the objects on which VM(M) instructions operate. They are also used as the parameter mechanism for procedures. Thus, the same construction is used to provide the arguments for VM(M) instructions and for procedure parameters. The operand stack is central to all levels of the MINT language.

4.2.2 Instruction Operation

The instructions of the VM(M) Virtual Machine operate entirely in terms of objects obtained on a stack. There are five basic instructions which reference Virtual Storage. These are used to obtain objects on the operand stack, to store the top object on the operand stack into Virtual Storage, or to increase by one the value in a Virtual Storage location. The string manipulation instructions also reference Virtual Storage based on the information on the operand stack. All other instructions manipulate the objects on the stack without referencing Virtual Storage. This instruction organization tends to minimize the frequency of Virtual Storage references. In addition, it minimizes the number of points at which Virtual Storage references may occur. This facilitates control over the addressing of, and access to, Virtual Storage.

This form of machine instruction architecture is termed *zero-address* architecture, as the instructions do not contain storage addresses. Using this architecture, a section of object program to add the two quantities a and b would appear as:

- Obtain the object a
- Obtain the object b
- Perform the addition operation.

The operation of addition is logically performed in the Virtual Machine in the following manner:

- Pop the top item from the stack
- Pop the next item from the stack
- Add the two popped items
- Push the result onto the stack.

Hence the Virtual Machine operation of addition will remove two items from the stack and return one item which is the sum of the two removed items. The sequence:

- Obtain a
- Obtain b
- Add

is standard *reverse-Polish* notation. Using this zero-address architecture, items may be stacked to any level to effect a desired result. For example the expression $a + b * c$ is generally interpreted as meaning add a to the product of b and c. This is expressed in reverse Polish as:

1. obtain a
2. obtain b
3. obtain c
4. multiply
5. add

After operation (3) has been performed three objects are present on the stack: c at the top, then b and then a. After operation (4), which functions much as the addition operation, there are two objects on the stack: The quantity a at the bottom, and the quantity resulting from the operation $b * c$ at the top. The add operation (5), as previously described, removes these two objects, sums them, and returns to the stack the resulting object whose value is $a + b * c$.

A basic function of the compiler is to generate from source expressions the reverse-Polish sequences which are required for evaluation by the Virtual Machine.

4.3 Compiler Operation

The compiler translates source text into VM(M) object text. MINT source text may be written at various levels. Thus, the work done by the compiler is also variable. If the source text is at the lowest level, and thus contains only simple data variables, constants, and VM(M) instructions, the compiler only assembles the instructions in correct sequence and assigns and sets storage addresses and contents. If higher level constructs are used, the compiler references directives or macros as the names of these are identified in the input stream in order to modify the translation result or to modify the actual input text. Eventually, the input is reduced to the lowest level form. This is then translated into VM(M) object text in Virtual Memory. Thus, the compiler may be made to operate in a manner, and at a level, similar to a conventional assembler, or it may be made to operate at a very high level where objects with new or complex structure are defined and referenced. The structure of the MINT system tends to encourage high level use.

4.4 The Dictionary Facilities

The dictionary system determines all MINT behavior. Manipulation of the list of dictionaries provides flexibility in the transformation of information. An important example of the use of dictionaries is their use in “auto-

compiling" the MINT compiler itself. The dictionary system starts with the use of the class DICT.

4.4.1 The CLASS DICT

Class DICT is used to introduce a new dictionary. Its form is:

```
DICT <dictionary name>
```

After introduction the dictionary is set by:

```
<dictionary name>: HDICT
```

where HDICT is a macro that builds the data structure required for a dictionary. Typically, a dictionary will be introduced and set by, for example:

```
DICT DC1: HDICT
```

After a dictionary name has been introduced and set it may be referenced, much like a DIR. Referencing a dictionary name causes that dictionary to become the current *active* dictionary. If the dictionary had not previously been referenced, it is initialized and pushed onto the dictionary list (see below). The compiler's dictionaries, MAINDIC and INTDIC, may be referenced in this way. However, it is recommended that the user introduce and use his own dictionaries. Thus,

```
DICT DC1: HDICT
DICT DC2: HDICT
.
. text section 1
.
DC1
.
. text section 2
.
DC2
.
. text section 3
.
DC1
```

will have the following effects:

1. The two new dictionaries DC1 and DC2 are created. Within text section 1 new introductions are made into the dictionary that was previously active.

2. Within text section 2 new introductions go into dictionary DC1.
3. Within text section 3 new introductions go into dictionary DC2.
4. After the last line (DC1) operation continues using DC1 for introductions.

4.4.2 The \ and % Operators

Two operators permit *transient* use of dictionaries, i.e. the dictionary name which follows the operator is used once and then the dictionary structure is returned to its previous state. The two operators are \ which causes identifier lookup in the specified dictionary, and % which introduces an identifier into the specified dictionary. Thus,

```
DICT DC2: HDICT
DICT DC4: HDICT
          DC2
VAR DCVAR:25
          DC4
VAR DCVAR:50
FN FF: ENTRY
          \DC2 DCVAR ->@TEMP
          EXIT
```

is a function which would generate text to obtain the the value of DCVAR in dictionary DC2 even though dictionary DC4 would normally have been searched first. The % operator directs introduction to the named dictionary, as, for example:

```
%DC4 VAR DCVARX:75
```

would introduce the variable DCVARX into dictionary DC4 regardless of what dictionary was currently active.

4.4.3 The SETDIC Function

This function provides a means of setting a dictionary's access control. Its general form is:

```
SETDIC(<value>, <dictionary address>)
```

The function sets <value> as the access control for the dictionary whose address is given. At present the following access controls are available:

Value	Meaning
0	dictionary is not used for lookup (locked)
1	dictionary may be read or written
2	dictionary is read-only (lookup only)

Thus, for example: SETDIC(0, @INTDIC) will exclude INTDIC from searches, and SETDIC(1, @INTDIC) will return it to read/write state so that it will be searched. For some purposes it is more convenient to lock a dictionary rather than to remove it from the dictionary chain. SETDIC may be applied to any dictionary regardless of whether it is in the dictionary list. New dictionaries are always initialized as read/write.

4.4.4 The LOCK, RD-ONLY, and UNLOCK Directives

These three directives reference NEXTELT, then SETDIC to set the requested dictionary state. Each requires a dictionary name as its argument. LOCK makes the dictionary entries no longer visible, UNLOCK makes them visible, and RD-ONLY prevents the dictionary from being updated. UNLOCK INTDIC is the replacement for ICL$ and LOCK INTDIC is the replacement for RCL$.

4.4.5 The LASTDIC Function

This function provides a means of setting the point at which the search of the dictionary list should be terminated. LASTDIC expects the top item on the stack to be the address of a dictionary in the dictionary list. This pointer is saved so that subsequent dictionary list searches stop after the dictionary whose address was provided.

4.4.6 The ICL$ and RCL$ Directives

The RCL$ directive performs a SETDIC(0, @INTDIC), and ICL$ performs a SETDIC(1, @INTDIC). Thus, RCL$ is equivalent to LOCK INTDIC and ICL$ is equivalent to UNLOCK INTDIC. These directives are included to maintain compatibility with MINT-2.

4.4.7 Notes on Dictionary Manipulation

There are two important points to keep in mind when managing multiple

dictionaries. First, each dictionary record contains a pointer to the dictionary record for its CLASS. Therefore, items of class CLASS should not be introduced into dictionaries which are then subsequently removed if identifiers of that class are also introduced into other dictionaries which are retained. There is no check that a CLASS pointer still points to the intended dictionary item. Second, if multiple introductions of the same identifier are made it is important to ensure that the point at which the identifier is inserted and the dictionary search order are such that the intended identifier is found. A prominent situation in which this could be a problem is the introduction of identifiers which are the same as ones in INTDIC after an UNLOCK INTDIC directive. The new identifier will be inserted in MAINDIC, which is searched after INTDIC.

When the compiler is initialized it creates and references a dictionary named USERDIC. Unless there is some special reason to do so, it is better not to introduce new identifiers into the compiler dictionaries, MAINDIC and INTDIC.

4.4.8 The Dictionary List

The dictionaries that are in current use are members of the list whose list pointer is ENVLIST. Dictionary addresses may be pushed and popped from this list either directly or by using the directives described below. In addition there is a pointer to an item in the dictionary list which determines the dictionary into which new definitions are added. Initially, this pointer points to the top item in the list.

The dictionary structure after compiler initialization is:

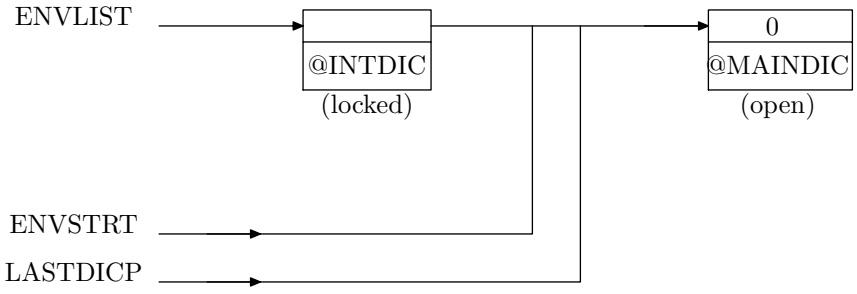


Figure 4-1. Initial Dictionary List

The items in Figure 4-1 are defined as follows:

ENVLIST	Pointer to start of dictionary list. All dictionary searches start at the dictionary pointed to by ENVLIST.
ENVSTRT	Pointer to current *active* dictionary. All new definitions are inserted into this dictionary.
LASTDICP	Pointer to the *last* dictionary to be searched. Setting LASTDICP to a dictionary before the last in the list permits searches of "windows" of dictionaries.

4.4.9 BLOCK and ENDBLOCK

The BLOCK directive performs the following operations:

1. A new dictionary is allocated, initialized to be empty, and pushed onto the dictionary list.
2. The current active dictionary pointer is saved on a stack and the active dictionary is set to the top of the dictionary stack.
3. The address of the new dictionary is saved in an internal list.

The ENDBLOCK directive performs the following operations:

1. The top address is obtained from the list used in BLOCK. This dictionary is removed from the dictionary list. Note that the last dictionary created by a BLOCK directory is the one removed regardless of any dictionaries that may have been pushed onto the dictionary list by other means.
2. The top item on the active pointer stack is obtained and the active dictionary pointer is reset as it was before the previous BLOCK directive.
3. The record space used by the dictionary records and dictionary index table is released.

These actions have the effect that a BLOCK/ENDBLOCK sequence is transparent in the sense that the dictionary list and pointers are put back as they were before the BLOCK directive, but any non-BLOCK changes to the dictionary list are preserved. Thus, specifically, a dictionary may be added by PUSHNDIC or PUSHODIC after a BLOCK directive and it will remain in the dictionary list after the following ENDBLOCK. For example, if the dictionary list was as shown in Figure 4-1 and then a BLOCK directive was obeyed, the list would be:

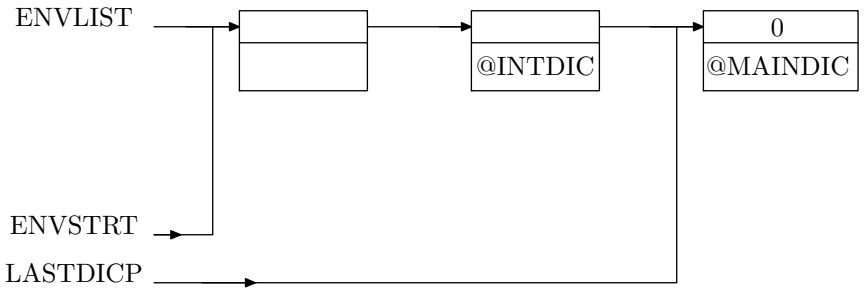


Figure 4-2. Dictionary List after BLOCK

4.4.10 SAVBLOCK and SETBLOCK

The SAVBLOCK function acts exactly like ENDBLOCK except that the dictionary address is stored at the address provided on the stack and the dictionary records are not released. The SETBLOCK function acts exactly like BLOCK except that the dictionary whose address is pointed to by the address on the stack is pushed, rather than pushing a newly initialized dictionary. Thus,

```
VAR SAVDIC:0
FN SAVRES:ENTRY, SAVBLOCK(@SAVDIC), SETBLOCK(@SAVDIC),
        EXIT
```

would perform an ENDBLOCK, but then save the dictionary address in SAVDIC, and perform a BLOCK, but restoring the dictionary as the new top item.

Note that SAVBLOCK does not "accumulate" dictionary records as was the case in previous versions of MINT. In MINT-2 a SAVBLOCK of a directory using the directory address of a directory which already contained entries would result in a merge of the old entries with the new entries. In MINT-3 SAVBLOCK simply saves the named directory.

4.4.11 Example use of UNLOCK INTDIC

The example below shows the list for the case where the compiler has been initialized and then an UNLOCK INTDIC directive has been obeyed.

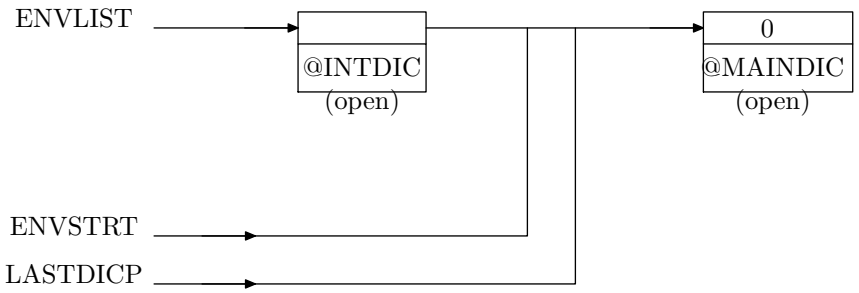


Figure 4-3. Dictionary List after UNLOCK INTDIC

The UNLOCK INTDIChas the effect that INTDIC and MAINDIC are searched when an identifier is matched, but new definitions are inserted in MAINDIC. Note that the requirement to match the longest string means that the entire dictionary list must be searched on all matches.

4.4.12 The Compiler Dictionaries

The dictionary MAINDIC is the base dictionary for the system. It contains the standard set of identifiers. Without this dictionary none of the normal MINT identifiers can be matched. While a POPUP(@ENVLIST) will pop this item if it is the only dictionary in the list, this is not a good idea in most cases. When the compiler is initialized the dictionary list pointers are set as shown in Figure 4-1.

4.4.13 Compiler Dictionary List Manipulation

4.4.13.1 PUSHNDIC

This function allocates a new dictionary, initializes it to an empty state, pushes its address onto the dictionary list, and returns the address on the stack. Note that PUSHNDIC does not modify ENVSTRT.

4.4.13.2 PUSHODIC

This function expects a dictionary address on the stack. It pushes this

address onto the dictionary list. Note that PUSHODIC does not modify ENVSTRT.

4.4.13.3 POPDIC

This function pops the top dictionary address from the dictionary list and releases the dictionary and record space. Note that POPDIC does not modify ENVSTRT.

4.4.13.4 ACTDIC

This function sets the active dictionary pointer, ENVSTRT, to the dictionary whose address is supplied on the stack. Identifiers are always introduced into the active dictionary.

4.4.14 Definition of Dictionary Records

Figure 4-4 shows the structure and contents of the dictionary records. These records are the core of the system. They can be changed as needed, but this should be done with care in order not to prevent correct operation of the existing compiler functions.

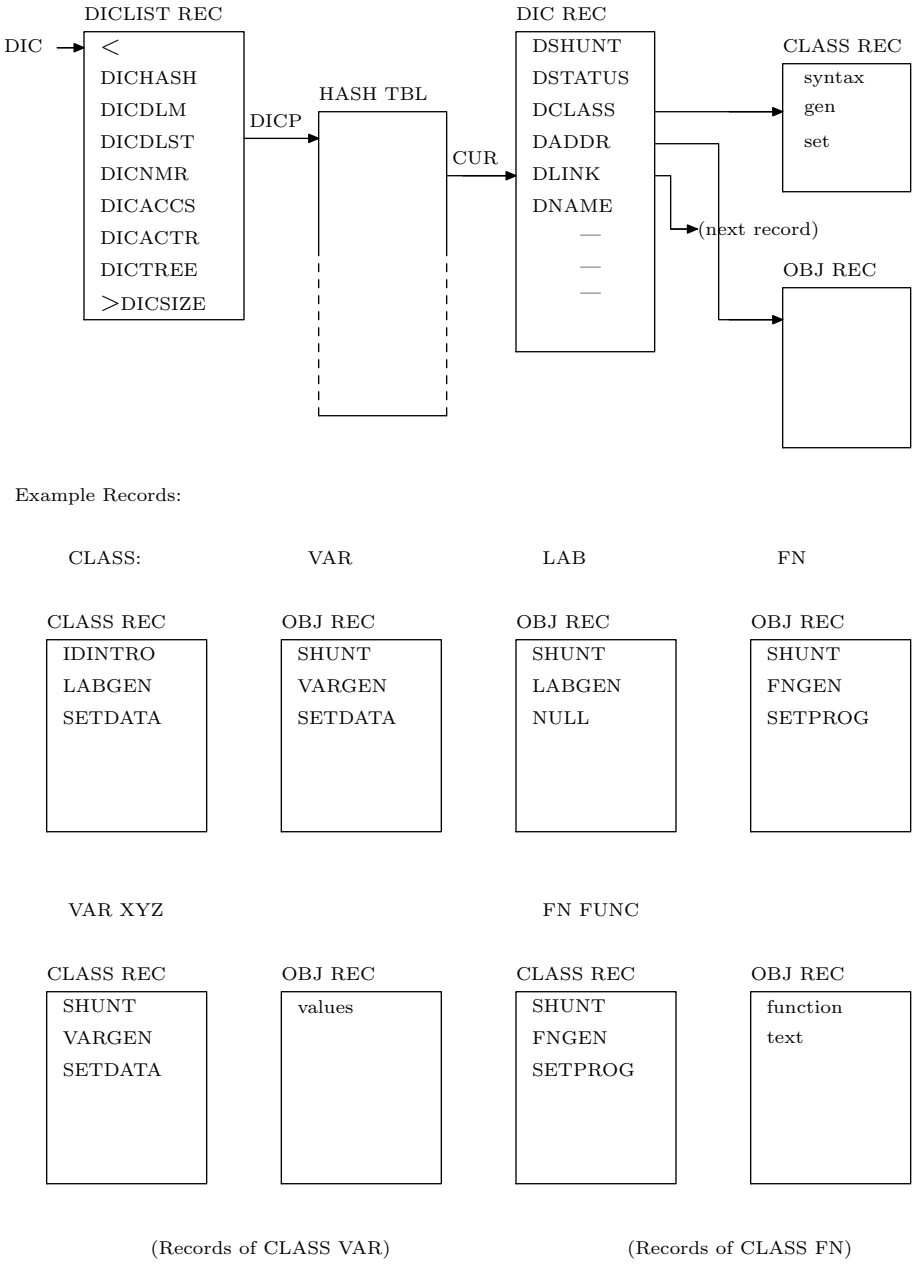


Figure 4-4. Dictionary Record Structure

4.4.15 Dictionary and Identifier Displays

The LV$ directive has been modified so that it displays the identifiers in each of the dictionaries in the current list from ENVLIST through LASTDICP. It also lists the dictionary name and count of items for each dictionary. The new directive, OBJ, provides a convenient means to display the identifiers in a given dictionary. (see below.)

4.4.15.1 Display of Objects

The directive LISTDICS lists the name and item count for each dictionary in the current list from ENVLIST through LASTDICP. The directive OBJ <object name> lists the properties and body of any object whose CLASS is known within the compiler. Specifically, if the object is a FN or DIR the object text will be displayed, and if the object is a DICT, the identifiers in that dictionary will be displayed.

The directive CURDIC will display the identifiers in the currently active dictionary.

The directive PREVDIC will display the identifiers in the "previous" dictionary. One the first use of PREVDIC, after a use of CURDIC, it will display the contents of the dictionary previous to the current active dictionary. If PREVDIC is referenced again, it will display the contents of the next previous dictionary. Further references provide displays of further dictionaries until the end of the dictionary list, at which point the cycle will repeat.

4.5 Use of NOW and PDUMP

Note that since NOW references BLOCK and since BLOCK pushes a new dictionary onto the dictionary list, constructs like:

```
NOW PDUMP('SAVE'), GO 32768 !
```

are not a very good idea since an extra dictionary will have been left on the list due to the fact that the ENDBLOCK in ! is not executed. For this reason a directive, CDUMP, is available (See Section 6.6.5). CDUMP expects two arguments which are the filename for the PDUMP and the address to which to transfer on reload. If this address is 0 a normal EXIT is taken on reload. Thus, a standard means of creating a PDUMP of the compiler is:

```
CDUMP 'COMP' 32768
```

4.6 Auto-compilation Facilities

Compilation of the compiler by the compiler is accomplished through use of the dictionary manipulation facilities and a few utility routines contained in the file MINTAUTO.

4.6.1 The Dictionary List during Auto-compilation

The text in MINTAUTO introduces and sets two dictionaries, MAUTO and IAUTO. When the AUTO directive is referenced the active dictionary is set to MAUTO. IAUTO is referenced in the text when it is required to introduce identifiers in the compiler internal dictionary. Thus, for the auto-compilation process MAUTO and IAUTO correspond to MAINDIC and INTDIC respectively. The GENSYS directive copies and relocates these dictionaries into data space and converts them into MAINDIC and INTDIC.

4.6.2 OLDIC

During auto-compilation a variable, BASESTRT, is used to control the point in the dictionary list at which identifier searches are started. BASESTRT is set to point to INTDIC. OLDIC does a lookup starting at BASESTRT. This causes previously defined identifiers (introduced in INTDIC or MAINDIC) to be found through OLDIC lookups, but new identifiers to be inserted and matched in MAUTO or IAUTO.

4.7 Compiler States and Data Declarations

The following Sections explain the management of procedure-space and data-space, and the allocation of reserved storage.

4.7.1 States

During the compilation of MINT source text the compiler is in one of two states, namely *program* or *data*. In program state all source text is compiled into procedure-space in a form suitable for execution by the Virtual Machine. Program state is entered initially when the compiler is loaded, and also when an identifier of class FN or DIR is set by means of the :

operator. The compiler automatically maintains the current end address pointers for procedure-space and data-space.

Data state is set when an identifier of class VAR or MACRO is set by means of :. In this state source text is compiled into data-space and is usually non-executable. In data state, constants may be expressed and identifiers referenced. In the latter case the address of the identifier is compiled just as if the identifier name had been preceded by an at-sign (@). (See Section 2.5.2 for definition of the @ directive.) For example:

```
VAR x:0
```

introduces the variable x and sets its address value to the next available data location. The value 0 is compiled and generated into this data location. The pointer to the next available data location is incremented by one.

```
VAR cde: 26 49 51
```

introduces and sets the identifier cde and compiles three consecutive values 26, 49 and 51 beginning at the address assigned to cde. The data pointer is incremented by three.

```
VAR table: x cde 0 18 &(2*(a-b))
```

introduces and sets the variable table and stores in successive locations the addresses of the identifiers x and cde, the values 0 and 18, and a constant whose value is the evaluation of the expression 2*(a-b) (See Section 2.5.4 for constant evaluation rules).

```
VAR text: 'This is text.'
```

introduces and sets the variable text and stores the string

```
This is text.
```

The compiler is usually able to select its state automatically. It is possible to set the state explicitly by the directives PROG and DATA. These directives may be used when it is desired to change the state and the appropriate class of identifier is not being set at the time.

4.7.2 Data-space and Procedure-space

The default arrangement of data-space and program space is shown in Figure 1-2. The sizes of these spaces are controlled by MAXDS$ and MAXPS$, which are shown in Table 13-1. The default values set in these variables are described in Section 14.3. MAXDS$ and MAXPS$ are used to set MAX-

PLOC and MAXDLOC when the compiler is initialized.

If other areas of the virtual memory are to be used, the start address should be set in PLOC or DLOC and the upper limit address should be set in MAXPLOC and MAXDLOC. After PLOC and DLOC are reset, further allocations will take place based on the new addresses.

The total size of VSTORE is set by the VM load routine in the variable MAXVS$. The value is in units of 1024 VSTORE words. This information can be used to decide on the use of VSTORE above the MAXDS$ and MAXPS$ limits.

A directive, VSTOREMAP, may be used to display the layout of VSTORE. It provides a display of the form:

```
VSTORE Usage:
Reserved VSTORE (0-80):
          Identifier     VSTORE Location
          MAXDS$         00001
          MAXPS$         00002
          IDLOC$         00003
          CONT$          00004
          SYSDAT$        00005
          MAXVS$         00008
          EXOPT$         00009
          DREM           00011
          DATE           00012
          SYSID$         00021

          VSTORE through location 80 is reserved.
System information:
          SYSDAT$: 060709
          MAXVS$:   16384
          DATE:    060709
          SYSID$:  3.0
Resources used:
          DLOC:   04356, MAXDLOC:   32768
          PLOC:   40351, MAXPLOC:   65535
Record POOL from    523264 TO    458752
Item   POOL from    458752 TO    393216
```

4.7.3 Reserving Storage

Storage locations may be reserved by means of the RESERVE directive

whose form is

RESERVE IPAR-expression .

This directive has the effect of reserving the number of locations specified by the evaluation of the IPAR-expression. The expression may be any valid MINT expression as defined by the IPAR-expression mechanism which is explained in Section 8.2. For example:

```
RESERVE 30
RESERVE (2*nents)
```

are valid uses of RESERVE. In the first case 30 words of storage are reserved, while in the second case the number of words reserved is determined by the evaluation of (2∗nents). The RESERVE directive does not initialize the reserved space. Therefore, such space must be initialized by the user.

4.8 MINT Expressions

Three distinct forms of expressions are recognized in MINT:

- Arithmetic expressions
- Boolean expressions
- Address expressions.

4.8.1 Arithmetic Expressions

An arithmetic expression is a sequence of MINT identifiers, constants, arithmetic, logical shift, or binary operators which has the effect of generating object text which obtains an integer object. The arithmetic and binary operators are:

+	integer addition
−	integer subtraction
*	integer multiplication
/	integer division
`-->`	right shift logical
`<--`	left shift logical
NEG	integer negation
MASK	binary and
UNION	binary inclusive or

DIFFER	binary exclusive or
COMPL	binary ones complement.

Thus:

$$a + b - c$$

is an arithmetic expression. As in many high level languages, the operators * and / take precedence over + and −, so that the expression

$$a + b * c$$

will result in the production of the product of b and c, added to a. This precedence order, which is fully explained in Section 4.9, may be overridden by use of parentheses. Thus,

$$(a + b) * c$$

has the effect of summing a and b and then multiplying the result by c.

The logical shift operators expect two operands, first a word which is treated as a 16-bit logical quantity and second the shift count. Thus,

```
a--> 4
```

will perform a logical right shift of the bits in the variable a by 4 bit positions. The result is left on the operand stack. Note that these two operator names are composed of two minus characters and > or <. The logical shift operators have the same binding power as the binary operators MASK, UNION and DIFFER.

The unary operator NEG has the effect of negating an integer object. Thus,

```
NEG (a),
```

obtains the negative of the value of the variable a. The NEG operator has the same high binding power as other functions. (See Section 4.9 for a full description of binding and precedence.)

The binary operators MASK, UNION, and DIFFER each act on two operands to produce a single binary result. Thus,

```
9 MASK 5
```

yields 4.

The COMPL operator functions like the NEG operator, but it yields the complement of its single argument. It has the same binding power as other functions.

The current MINT system does not carry out type checking on arithmetic operands. Care should be exercised to avoid constructions which would have unintended type-inconsistency effects.

4.8.2 Boolean Expressions

A Boolean expression is a sequence of MINT elements whose effect is to generate object text which obtains a Boolean object. It consists of object obtaining expressions, relational operators and logical operators. The relational operators, which may be applied to any kind of object, are:

EQ	equal
NE	not equal
LT	less than
LE	less than or equal
GT	greater than
GE	greater than or equal.

For example if the variable a has a value of 10 and the variable b has the value 8 then:

a EQ b	yields false
a NE b	yields true
a GT b	yields true
a LE b	yields false.

Logical operators should only be applied to Boolean objects. The logical operators are:

NOT	logical complement
AND	and
OR	inclusive or
XOR	exclusive or.

The NOT operator obtains the logical complement of a Boolean object. Thus, if the expression

```
a EQ b
```

yields a true result, the expression

```
NOT (a EQ b)
```

will yield a false result. An example Boolean expression is:

```
(a EQ b) AND (c GT d) .
```

This expression will obtain a true result if both a equals b and c is greater than d, while

```
(xyz GT pqr) OR (g LT 0)
```

yields a Boolean true if either xyz is greater than pqr or if g is less than zero.

4.8.3 Address Expressions

An address expression is a series of MINT elements whose effect is to generate object text which obtains an address. It consists of integer objects, address objects and the address operators FROM and ADIFF. An address object is obtained from an address constant (@ directive) or from a reference to a label. The address operator FROM obtains an address from an integer displacement and an address. Thus,

```
3 FROM @table
```

is an address obtaining expression which obtains on the stack the address of the fourth location of table. Negative displacements are effected by negative integers. For example,

```
MINUS 3 FROM @buffer
NEG(a+b) FROM @list .
```

The address operator ADIFF obtains the difference between two addresses. Thus, for example, the length of a table may be obtained by:

```
@tabend ADIFF @table .
```

4.9 Precedence

The reordering of the identifiers which compose expressions is determined by the *precedence* associated with each identifier. It is conventional to refer to identifiers with low precedence numbers as having high binding power. This means that the identifiers with low precedence tend to stay *bound* in their position whereas identifiers with higher precedence tend to be carried along within an expression. The intent of the precedence scheme is to allow use of natural ordering for readability, but ensure the required reverse-Polish sequencing in the resulting object text. In order to see how reordering operates consider the expression:

```
a + b * c -> @d
```

where a, b, c, and d are variables, + and * are the add and multiply operators, − > is the store operator (See Section 6.2.4), and the @ directive generates an address constant (See Section 2.5.2). The required reverse-Polish sequence is:

```
obtain a
obtain b
obtain c
multiply
add
obtain @d
store .
```

Thus, the required reordering results in the sequence:

```
a b c * + @d -> .
```

The reordering which generates reverse-Polish sequences is accomplished by the compiler *shunt* procedure. As indicated in Section 2.2, syntax action is performed when an identifier is encountered on the input stream. The syntax procedure applies the shunt procedure. The shunt procedure causes the dictionary item associated with the identifier to be linked at the top of a list of items for which generative action is pending. This list is referred to as the *shunt stack*. The generative procedure is called by the shunt procedure when a dictionary item is to be removed from the shunt stack. The shunt procedure causes the reordering of the input stream by requiring that each new item to be linked onto the shunt stack have the lowest precedence number of all items currently on the active shunt stack. If any items exist on the stack which have a lower or equal precedence number, these items are removed. This causes generative action to be performed for these items. There are two identifiers which modify this basic logic. When an opening parenthesis is encountered it is unconditionally pushed onto the shunt stack and is given a precedence number of 1000. In addition, an opening parenthesis on the shunt stack is treated as the current stack bottom. Thus, input text following an opening parenthesis is processed separately from the preceding text. When a closing parenthesis is encountered all items with precedence lower than or equal to 99 are removed. This ensures that all items up to the last opening parenthesis are removed. In addition, the closing parenthesis causes removal, without any generative action, of the opening parenthesis. This causes the items which were below the opening parenthesis to become available for removal. The operation of this procedure on the expression given above is shown in Figure 4-5.

Input	**Shunt Stack**	**Precedence**	Generative Action
a	a	0	
+	+	10	obtain a
b	b	0	
*	*	8	obtain b
	+	10	
c	c	0	
	*	8	
− >	− >	22	obtain c multiply
@d	@d	0	
	− >	22	

Figure 4-5. Example Shunt Operation

The Figure shows the state of the shunt stack and any generative actions after each identifier has been processed. Thus, for example, after the variable c has been processed the shunt stack contains c, ∗, and +. Since nothing was forced off of the shunt stack at this point, there was no generative action taken. The example shown in Figure 4-5 also shows that there may be items remaining on the shunt stack at the end of an expression. The comma directive, which has no generative action, has precedence number 99. Therefore, it may be used to force items off of the shunt stack. Comma is therefore frequently used as an expression terminator. If a comma is omitted, unintended floating of identifiers may occur. Comma and closing parenthesis cause exactly the same action except that the closing parenthesis also removes the opening parenthesis which marks the current stack bottom.

Figure 4-6 clarifies the operation of parentheses and comma, using the input sequence:

$$a + SUM(b, c*(d + e)) * f$$

where a, b, c, d, e, and f are variables and SUM is a function.

Input	**Shunt Stack**	**Precedence**	Generative Action
a	a	0	
+	+	10	obtain a
SUM	SUM	0	
	+	10	
(	(	1000	
	SUM	0	
	+	10	
b	b	0	
	(	1000	
	SUM	0	
	+	10	
,	(	1000	obtain b
	SUM	0	
	+	10	
c	c	0	
	(	1000	
	SUM	0	
	+	10	
*	*	8	obtain c
	(	1000	
	SUM	0	
	+	10	
(	(	1000	
	*	8	
	(	1000	
	SUM	0	
	+	10	
d	d	0	
	(	1000	
	*	8	
	(	1000	
	SUM	0	
	+	10	
+	+	10	obtain d
	(	1000	

	*	8	
	(	1000	
	SUM	0	
	+	10	
e	e	0	
	+	10	
	(	1000	
	*	8	
	(	1000	
	SUM	0	
	+	10	
)	*	8	obtain a
	(	1000	add
	SUM	0	
	+	10	
)	SUM	0	multiply
	+	10	
*	*	8	SUM
	+	10	
f	f	0	
	*	8	
	+	10	
,			obtain f multiply add

Figure 4-6. Action of Parentheses in Shunt Operation

4.9.1 Precedence Numbers

Table 4-1 gives the precedence numbers for all defined identifiers. These are the numbers shown in Figures 4-5 and 4-6. EXIT causes a special compiler action which prevents it from floating. It is therefore not necessary to place a comma after the EXIT operation.

Table 4-1. Precedence Number Assignment

Precedence number	Identifier
0	all functions, directives, variables, constants, labels, and operators other than those listed below
4	MASK UNION DIFFER −− >< −−
8	* /
10	+ - FROM ADIFF
12	EQ NE LT GT LE GE
14	NOT
16	AND
18	OR XOR
20	CHOOSE
22	− > <=> YES NO GO DO
24	ENTRY EXIT
99	,)
1000	(

4.9.2 Precedence Manipulation (PRIORITY)

The precedence or shunt factor of each identifier has been moved from the CLASS record to the identifier's dictionary record. Due to this change it is possible to set an operator's precedence independent of its CLASS. The directive PRIORITY provides the means of setting precedence. In principle, precedence could be changed dynamically during processing. The use of PRIORITY is as follows:

PRIORITY <n> <identifier introduction>

This statement causes precedence <n> to be applied to the following introduction. After the introduction the default priority is reset to 0. Thus, an example use is:

PRIORITY 4 PRIMOP MASK 15

This is the text used in the compiler to set priority 4 for the operator MASK, and set its value (operation code) to 15.

4.9.3 Use of Parentheses

It is commonly desirable to modify the natural order of the floating process

described above. As has been seen, the expression:

a + b ∗ c ,

yields

a b c ∗ +

as the multiply operation has a lower precedence number than addition. The effective order of operations can be modified by enclosing a + b in parentheses. Any expression in parentheses is sorted entirely on its own and independently of expressions outside the parentheses. The effect of the parentheses is to temporarily suspend the current sorting operation. The text within parentheses is sorted entirely on its own. Thus, the ordering of

(a + b) ∗c,

is

a b + c ∗

which evaluates a+b first and then multiplies the result by c. Parentheses may be nested to any depth as, for example:

a ∗(b + c ∗(d + e))

which yields

a b c d e + ∗ + ∗ .

Parentheses are also used following a procedure reference in order to delimit the parameters required by the procedure. Thus, for example, a function reference could be written as

PROD(X,Y),

which yields

X Y PROD .

4.9.4 Use of Comma

Because of the syntactic freedom of the MINT language there is no such thing as a statement. In MINT programming it is frequently necessary to indicate the termination of certain text sequences in order that the correct reverse-Polish sequences can be formed. This is effected by means of the comma (,) directive, since this directive has higher precedence than any other operators except the parentheses. While it is not usually necessary to remember the reordering process in detail, it is important to be aware that

many operators are floating through an expression, and that the process is terminated by a comma. Thus,

```
a + b -> @c ,
p * c + 3 -> @r ,
```

is an illustration of the correct use of comma. If the comma in the first line were omitted then the first - would float up to the second one, with unintended results.

4.9.5 Implied Commas

The compiler facilities for iteration and conditional expressions have commas included in order to avoid the possible floating of operators into the beginning or out of the end of these expressions. The following compiler identifiers, which are discussed in Chapter 6, result in implied commas:

```
THEN
ELSE
<
>
WHILE
START
REPEAT
```

Additionally the comma action is performed whenever an identifier is set, with the exception of labels (class LAB).

4.10 Problems

4.10.1 Problem 1

Draw a diagram showing VSTORE, the stacks, stack pointers DLOC and PLOC, and locations of object text after compilation of the text:

```
LAB step1: .
```

4.10.2 Problem 2

Use a MINT system with LIST LOCS LCODE set in order to investigate the object text which is created for sample source sequences such as:

```
(a + b) * c,
MINUS 3 FROM @buffer .
```

4.10.3 Problem 3

Verify the precedence ordering of the expression:

```
a + b * c -> @d .
```

4.10.4 Problem 4

Verify the precedence ordering of the sequence:

```
a + b -> @c
p * c + 3 -> @r
```

and examine the effect of a comma at the end of the first line.

5. The Macro Facility

5.1 Introduction

MINT macros provide a general means of text replacement. A macro reference simply causes replacement of the reference by the macro text. There are no special indicators for references or parameters. Thus, any string may become a macro reference, and any string may be used as macro text. Macro references may be nested to any level.

5.2 Macro Bodies

A macro is an identifier that has been introduced as CLASS MACRO, and is set to a string constant which is the macro body. Thus, the macro x is introduced and set by:

```
MACRO x:'This is a macro-body' .
```

Any reference to the identifier x directs the compiler to the string, which is then processed as normal input. At the end of the string, input reverts to the text following the reference.

5.3 Macro Parameters

Due to the flexibility of MINT expression structure, and the lack of predefined syntactic entities, there is no fixed syntax to indicate parameter references. The simplest means of achieving the effect of conventional parameter substitution is through the definition of further macros. Thus,

```
MACRO x:' SUM(A, param1), OPINT '
MACRO param1: 'B*C'
```

would result in compilation of the string

```
SUM(A, B*C), OPINT
```

when x is referenced.

5.4 MINT System Macros

Several standard MINT constructs are implemented by means of macros. Two examples of this usage are given here. Conditional expressions and iteration are also implemented by means of macros. These constructs are discussed in Chapter 6.

5.4.1 Forward Referencing

The < and > macros provide an *anonymous* forward referencing mechanism. For example the sequence:

```
LAB lb10
        :
GO lb10,
        :
lb10:
FORGET lb10
```

may be more conveniently written as:

```
        :
GO      <
        :
        >
```

This is because the < and > macros are defined as:

```
MACRO <:' LAB OUT$ OUT$,'
MACRO >:',OUT$:FORGET OUT$ '
```

where the < macro introduces the label OUT$ and references it while the > macro sets the label and then forgets it. This facility may usefully be employed in data state. For example,

```
VAR linklist:<1><2><3>0, 4,
```

compiles a linked list of the items 1, 2, 3, and 4. Figure 5-1 shows the resulting list structure. See also Chapter 9 for more detailed discussion of lists.

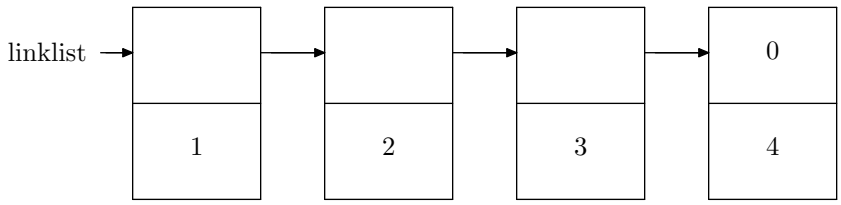


Figure 5-1. Linked List Constructed with <>

Note that the label OUT$ may be referenced within the <. . .> construction.

5.4.2 Backward Referencing

A facility analogous to the < and > forward referencing mechanism is provided by the WHILE and BACK macros for backward referencing. These macros are defined as:

```
MACRO WHILE:',LAB LOOP$:'
MACRO BACK:'LOOP$, FORGET LOOP$'
```

Thus a loop may be written as:

```
WHILE
        :
        :
GO BACK,
```

The label LOOP$ may be referenced within a WHILE . . . BACK construction.

5.5 Some Additional Macros

In this Section we describe four simple macros which have been found generally useful. These can easily be included in a standard system using the techniques described in Sections 12.2, or 14.6 and 14.7.

5.5.1 Loop Iteration

It is often required to iterate a given sequence a predetermined number of times. The following two macros facilitate this kind of iteration using the

following construction:

```
count TIMES text ... LOOP
```

where:

count	is an expression which obtains the desired iteration count,
TIMES	is a macro which provides the WHILE ... START sequence,
text...	is the sequence which is to be iterated,
LOOP	is a macro which terminates the WHILE loop.

An example use of this iteration construction is:

```
5 TIMES ADVCH LOOP.
```

This text would have the effect of advancing the compiler input character pointer (See Section 10.5.8) 5 times.

The text for the two macros is as follows:

```
MACRO TIMES:' WHILE -1, DUP GE 0 START '
MACRO LOOP :' REPEAT, LOSE, '
```

5.5.2 Abbreviated ENTRY and EXIT

Some people find it tiring to type ENTRY and EXIT when writing many short procedures. For such people, the following two macros may be convenient:

```
MACRO { : 'ENTRY,'
MACRO } : 'EXIT'
```

5.6 Problems

5.6.1 Problem 1

Compare the results of the use of normal source text, and the use of MACRO's.

6. Basic MINT Constructs

6.1 Introduction

This Chapter introduces the MINT constructs which reference VSTORE, manipulate objects on the operand stack, and which provide control of instruction execution.

6.2 VSTORE Referencing

There are five basic operators which reference VSTORE. Three of these (GET, GETV, and VAL) obtain values from VSTORE. The -> (store) operator stores a value at a specified VSTORE location. The operator ADV increments by one the content of a VSTORE location. The external interface operators (INCH and OPCH) and the string operators (GETCH, PUTCH, MATCH, and DICMATCH) also reference VSTORE. These operators are discussed in Chapter 10. All other operators operate only on stack objects.

6.2.1 Obtaining Objects (GET, GETV)

The most common operation in program text is the obtaining of objects on the operand stack. Therefore, the MINT compiler provides a uniform and simple means of expressing this operation for various objects. Whenever an identifier or a literal is recognized in the compiler input stream, object text is generated which obtains the value of the referenced object. Thus, the literal reference:

```
4926
```

will cause the compiler to generate text to obtain the integer value 4926. Or, after introduction and setting by:

```
VAR abc:24
```

a reference to abc will generate text which obtains the value of the variable abc, i.e. 24. The same rule applies to integer constants (introduced by ICON). Similarly, a reference to a label (LAB) results in text to obtain

the address value of the label. The primitives which are generated for these operations are GET, which obtains a constant value, and GETV, which obtains the value of a variable. Since the occurrence of a literal or identifier name causes the compiler to generate a GET or GETV using the required address, it is never necessary to directly use these primitives. Since constants cannot be modified they are generated as part of the instruction text. The GET primitive obtains the constant value from the word immediately following the GET instruction. GETV, on the other hand, uses the value which follows the GETV as the address (generally the address of a variable object) of the VSTORE location from which it obtains the value.

6.2.2 Indirect Addressing (VAL)

Indirect addressing is effected by means of the VAL operator. Its form is

VAL (address-obtaining-expression) .

This has the effect of obtaining the contents of the obtained address. Thus, if table is a variable assigned to the start of a table, and xyz is a variable, then

```
@table -> @xyz,
VAL(xyz)
```

will obtain the contents of the first location of table, as xyz is an address obtaining expression yielding the address of table. Similarly

VAL(3 FROM xyz)

obtains the contents of the fourth location of table. If ABC is a variable,

```
    ABC
and VAL(@ABC)
```

are equivalent.

Indirect addressing may be carried out to any level. Consider in the above examples that table itself contains a list of addresses. Then the construct

VAL(VAL(1 FROM xyz))

would obtain the contents of the location pointed to by the second location of table. The sequence of operations is:

1 FROM xyz

obtains the address of the second location of table. The inner VAL operator obtains the contents of that location. The outer VAL operator treats the obtained value as an address and obtains its contents.

6.2.3 The ADV Operator

Since incrementation by one is a very common operation, a primitive is provided to make the writing of the operation more compact and to allow faster execution by the Virtual Machine. This operator, ADV, expects as its single argument the address of the object which is to be incremented by one. Thus,

```
ADV(@xyz)
```

yields exactly the same result as

```
xyz + 1 -> @xyz .
```

6.2.4 The -> Operator

The `->` (store) operator is used to store an object into a Virtual Storage location. The store operator removes the top two objects from the stack. It takes the top object as the Virtual Storage address, and stores the second object at that address. The usual `->` construction is:

```
-> address-obtaining-expression .
```

Therefore to store the value 3 into a variable xyz the construction

```
3 -> @xyz
```

may be used, where @xyz is an address obtaining expression yielding the address of the variable xyz. Of course,

```
3 @xyz ->
```

is an equivalent expression. The destination for the store operation must be, in effect, an address obtaining expression. If the construction

```
3 -> xyz
```

had been used the value of the variable xyz would be obtained instead of the address. This value would be treated as an address. Such a construct is quite valid where a variable is being used as an address pointer. Consider:

```
@table -> @xyz .
```

This has the effect of storing the address of table in the variable xyz. Subsequently

```
10 -> xyz
```

will result in 10 being stored in the first location of table, as the obtained value of xyz is in fact the address of table. Note that

```
10 -> 3 FROM xyz
```

would result in 10 being stored in the fourth location of table, 3 FROM xyz being an address obtaining expression.

An attempt to store into an address outside the VM(M) Virtual Storage limits results in a VM(M) Virtual Machine error.

6.3 Operand Stack Management

In many instances it is required to access the top item of the stack more than once, and in some instances to remove it. It is also frequently required to exchange the top two items on the stack. Three operators are provided for more efficient object text production in these cases. The use of these operators both reduces the number of generated instructions required for a given result, and reduces the number of Virtual Storage references when the object text is executed. The three operators are:

```
DUP     - duplicate the top item on the stack,
LOSE    - discard the top item on the stack,
<=>     - exchange the top two items on the stack.
```

6.3.1 The DUP Operator

The DUP operator obtains on the stack a duplicate copy of the current top item. It is a convenient way of referring to and at the same time retaining the top item. For example if it is required to store the value zero into 3 variables p, q and r the construction:

```
0,                  . obtain zero
DUP -> @p,
DUP -> @q,
-> @r,
```

results in more efficient object text than writing:

```
0 -> @p,
0 -> @q,
0 -> @r .
```

As another example the absolute value of an integer object may be obtained by:

```
DUP LT 0 THEN <NEG>
```

(The THEN construction should be self-explanatory in this context. It is presented fully in Section 6.5.2.2.) In the example, the DUP operation results in two copies of the integer object. The top copy is tested for negative leaving the second copy to be negated or not depending on the result of the test.

6.3.2 The LOSE Operator

The LOSE operator discards the top item on the stack. An example of the use of LOSE is:

```
DUP GT MAX THEN <LOSE, MAX>
```

This sequence has the effect of testing if the top of stack value is greater than MAX and if it is the value is replaced by MAX. A frequent application of LOSE occurs in connection with the repeated DUP of an object which is used within a loop (See Section 6.5). When the loop is complete the object may be discarded by means of a LOSE.

6.3.3 The <=> Operator

The <=> operator exchanges the top two items on the operand stack. It is logically equivalent to:

```
VAR sav1:0
VAR sav2:0
->@sav1
->@sav2
sav1
sav2 .
```

However, the <=> operation does not make any Virtual Storage references for its operands.

As an example, consider a sequence which references a function which returns two objects. It is then required to divide the second object by the top object. If the function name is factors,

```
factors, <=>, /
```

will perform the intended operation.

6.4 Control Transfer

Three operations are available to alter the flow of control in a program. One is an unconditional transfer, the other two are conditional. In well organized programs little need will be found for these control transfer operators. Normal logical control is fully provided for by conditional expressions and iteration (Section 6.5), and functions (Chapter 7). The control transfer operators are provided as they form the basis for the higher level logical control and iteration structures, and for special purposes such as initial program start, or entry to a section of object text to be tested.

6.4.1 Unconditional Transfer (GO)

Unconditional transfer is effected by means of the GO operator whose form is:

```
GO address-obtaining-expression .
```

The following are example uses of GO:

```
GO label
GO begin
```

where label and begin have been introduced as labels. A loop could be written as

```
LAB xyz:
        :
        text
        :
GO xyz .
```

The address obtaining expression may be as general as desired. For example, consider:

```
GO VAL(x FROM @table) .
```

In this example @table is assumed to be the address of the first of a list of addresses. The value of x is used as a displacement from the start address of table. The VAL operator then obtains the contents of the table entry x entries from @table, which is treated as the address for the GO.

6.4.2 Conditional Transfer (YES/NO)

A simple form of conditional transfer is provided by the YES and NO operators. These operators effectively apply the GO operator to an address-obtaining expression depending on the value of a Boolean expression. The form for the operators is:

Boolean-expression YES/NO address-expression .

The YES operator applies a GO to the value of the address-expression if the Boolean-expression is true. Otherwise, the address-expression object is discarded. The NO operator applies the GO if the Boolean-expression is false. Otherwise, the address-expression object is discarded. For example in:

p+1 GT q YES lab1,

if p+1 GT q is true, the YES operator effects a jump to lab1, otherwise execution of the object program continues normally.

6.5 Conditional Selection and Iteration

MINT provides a range of conditional selection and iteration constructs. At a low level simple object selection and conditional skips are provided, while at a higher level general conditional selection and iteration are available. The higher level constructs are implemented by means of macros which use the low level facilities.

6.5.1 Object Selection

The CHOOSE operator is used to form object selection expressions. Its form is:

Boolean-expression CHOOSE
(object-obtaining-expression, object-obtaining-expression)

The Boolean expression is evaluated and the resulting object is tested. If it is true the object obtained by the first object-obtaining-expression is left on

the operand stack. If it is false the object obtained by the second expression is left on the stack. For example, the larger of two integer numbers may be obtained by:

a GT b CHOOSE (a, b) .

The expression a GT b obtains a Boolean object. If the object value is true then a is left on the stack. If the value is false then b is left on the stack. The implementation of the CHOOSE operator is such that all three of the parameters are obtained, then the selection is made. Thus, if any parameter is an expression the expression is always evaluated even if the parameter is not selected.

The CHOOSE operator may be combined with the GO operator to produce conditional transfers as, for example:

GO p EQ q CHOOSE(lab1, lab2) .

If p EQ q is true the GO will be applied to lab1. Otherwise, the GO is applied to lab2.

6.5.2 Conditional Execution

There are two forms for conditional execution. One form conditionally skips the next VM(M) instruction. The second form conditionally selects text sequences of any length. The first form is primarily intended for use where highly efficient simple branches are required. The second form is logically more general than the first.

6.5.2.1 The Skip Operators (TRUE/FALSE)

The skip operators are used to conditionally skip the VM(M) instruction which immediately follows the operator. The operator determines whether to skip by testing the top item on the operand stack. Thus, the operators take the form:

Boolean-expression TRUE

or

Boolean-expression FALSE .

The operator TRUE skips the next VM(M) instruction if the Boolean-expression obtains a false value. The operator FALSE skips if the Boolean-expression obtains a true value. If the operator does not skip, normal execution continues with the instruction following the operator. The Boolean-

expression is always evaluated, and the result is removed from the stack by the operator. For example:

```
n GT 4 TRUE EXIT
```

would have the effect that if n is greater than 4 the EXIT instruction is executed. Otherwise, the EXIT is skipped.

These operators should only be used for object text optimization where very highly efficient selection is demanded. The next Section describes the THEN construction, which is a more general-purpose substitute for TRUE/FALSE.

6.5.2.2 Conditional Expressions (THEN ... ELSE ...)

Conditional expressions allow the selection of alternate object text sequences depending on the value of a Boolean expression. The conditional expressions allow an optional ELSE clause. Thus, the two possible forms are:

```
Boolean-expression THEN <text>
```

or

```
Boolean-expression THEN <text ELSE text>
```

In the first form the text between the angle brackets is executed if the Boolean expression obtains a true value, otherwise it is skipped. In the second form the text following and preceding ELSE is executed if the Boolean expression obtains a true value. If the Boolean expression obtains a false value the text following the ELSE and preceding the is executed. Conditional expressions may be nested to any level. An example conditional expression is:

```
x GT 100 THEN <100 ->@x>
```

the value of x set to 100 if its value was greater than 100. To repeat the example in Section 6.3.1, the expression

```
DUP LT 0 THEN <NEG>
```

has the effect that if the duplicated item on the stack is negative it is negated, otherwise it is left as it is. As an example of the second form consider two variables a and b such that the lesser is required to be set equal to the greater of the two. The expression

```
a GT b THEN <a ->@b ELSE b ->@a>
```

will effect this because if a is greater than b it is stored into b, and if not then b is stored into a. As a further example, consider:

```
DUP GT max THEN <->@max ELSE LOSE> .
```

In this example the top item on the stack is duplicated and compared with the variable max. If it is greater the second copy is stored into max. If it is not then the second copy is discarded by means of the LOSE operator.

The conditional expressions are implemented by means of standard macros. These macros are:

```
MACRO <:        'LAB OUT$ OUT$,'
MACRO >:        ', OUT$:FORGET OUT$ '
MACRO THEN:     ', NO'
MACRO ELSE:     'LAB Z ,GO Z,>RENAME Z OUT$ '
```

Substitution of the above macro text into the example conditional expressions will yield the intended flow of control in terms of basic operators. Carrying out this substitution in a few sample cases will help to clarify the control structure, and helps to relate source text to object text.

6.5.3 Execution Iteration (WHILE ... REPEAT)

The general form of an iteration expression is:

```
WHILE Boolean-expression
START
        :
        text
        :
REPEAT
```

The iteration starts at the WHILE and the Boolean expression is evaluated. If the condition is true the section of text is executed and the repeat effects a jump back to the WHILE where the Boolean expression is evaluated once again. This process continues until the Boolean expression yields a false result. When this occurs the START effects a jump to the location just after the REPEAT.

Consider the following example to zero an array of 100 storage units:

```
0,                          . line 1
WHILE DUP LT 100            . line 2
START                       . line 3
```

```
    DUP FROM @array,          . line 4
    0, <=>, ->,               . line 5
    +1                        . line 6
REPEAT                        . line 7
LOSE                          . line 8
```

In line 1 the value zero is obtained on the stack. In line 2 the iteration begins with the Boolean expression which obtains another copy of the current top of stack value and compares it with the value 100. If the current value is less than 100, the iteration is executed. Line 3 indicates the start of the section of iterated text. In line 4 the current offset from the start of the array is computed. A copy of the offset is left on the stack. Line 5 obtains a 0 on the stack, exchanges the 0 and the current address in the array, and then stores the 0 at the current address. In line 6 the current offset is incremented by 1. Line 7 is the end of the iteration. From this point control returns to the WHILE in line 2. When the iteration is completed due to a Boolean false in the WHILE expression control passes to the expression after REPEAT, which is line 8. This line removes the offset value which was left on the stack.

Iterations may be written using only the conditional expressions, as for example:

```
LAB   out
LAB   loop:
          :
          text
          :
          exit-condition THEN <GO out<
          GO loop,
      out:
```

Alternatively, components of the general iteration expression may be used, as in:

```
WHILE
    :
    text
    :
    exit-condition NO BACK
continuation after loop.
```

The iteration expressions are implemented by use of standard macros. These are:

```
MACRO WHILE: ', LAB LOOP$:'
MACRO START: 'THEN<'
MACRO REPEAT: 'GO BACK>'
MACRO BACK: 'LOOP$, FORGET LOOP$'
```

6.6 Miscellaneous Constructs

There are several operators which provide useful service functions. The most important are the instruction intercept operator and the instruction emulation operator. Others include normal and error Virtual Machine stop operators, a host environment interface operator, an operator to save the current VSTORE, and a means of obtaining the current time value from the Virtual Machine. These operators are described below.

6.6.1 Virtual-Machine Instruction Intercept (TRAP)

The instruction intercept, or trap, operator is a key component of MINT analysis and diagnostic capabilities. The TRAP operator allows any program to gain control, as if by a procedure reference, just prior to each instruction execution. This allows information to be displayed, as in the T$ trace routine (See Section 8.6.2), operation usage data to be computed as in the instruction mix routines described in Section 12.7, or special conditions to be tested for as a part of diagnostic analysis.

The form of the TRAP operator is:

```
TRAP(address-parameter) .
```

The TRAP operator is used in two distinct instances:

1. to enable or disable instruction trapping, and
2. to return control to the main instruction path when the trap procedure has completed processing.

These two cases are distinguished by the location of the TRAP operator. If the TRAP operator is executed in normal text it is treated as an instance of the first case. If the operator is executed within the trap procedure then it is treated as the second case. Instruction trapping is enabled by the sequence

```
TRAP(address-parameter) .
```

When trapping is enabled the Virtual Machine will transfer control to the address given as the argument prior to execution of the next (normal) in-

struction. When control is transferred in this manner, the address of the next instruction is obtained on the operand stack, and further trapping action is suspended. The trap processing procedure may carry out any processing since it is a normal MINT procedure. When processing is complete, this procedure must return control by a reference to the TRAP operator of the form:

```
TRAP(address-parameter) .
```

The address-parameter must be the instruction address which was passed on the stack when the procedure was entered. When this trap instruction is executed the address-parameter is used to reset the next instruction address in the Virtual Machine, the next instruction is executed, and then trapping is re-enabled. When the sequence

```
TRAP(0)
```

is executed, instruction trapping is disabled.

The following example shows all of the steps which are involved in the use of trapping. If ST-TRAP is referenced, the text below will display the address of each subsequently executed instruction.

```
LAB TRFCN:                  . line 1
        OPINT(DUP), OPNL,   . line 2
        TRAP( )             . line 3
DIR ST-TRAP: ENTRY,         . line 4
        TRAP(@TRFCN)        . line 5
        EXIT                . line 6
DIR END-TRAP: ENTRY,        . line 7
        TRAP(0),            . line 8
        EXIT                . line 9
```

Lines 1 through 3 define the intercept procedure. Line 2 displays the address of the next instruction to be executed and line 3 terminates the trap procedure. Lines 4 through 7 define the directive ST-TRAP which initializes and enables trapping. Line 5 enables trapping and provides the address of the trap procedure. Lines 7 through 9 define the directive END-TRAP which turns off further trapping. Line 8 disables trap mode.

6.6.2 Virtual-Machine Instruction Emulation

The EMULATE operator provides a means of replacing the operation of any operation code by a standard MINT procedure. This permits selective analysis of the operation of individual primitives, the testing of possible new

primitives, or the investigation of specific effects based on the execution of a primitive. Emulation may be requested for any operation code (except 80 which is the operation code for EMULATE) including the currently undefined operation codes. The emulate operator requires two arguments; the operation code to be emulated and the address of the procedure which will be executed in place of the primitive. If the procedure address is zero emulation for the given operation code is discontinued and normal VM execution is resumed. The procedure which carries out the emulation must be written according to the following rules:

1. On entry to the procedure the operand stack contains the address of the instruction which is being emulated. The link stack has not been modified and therefore it may not be used for return of control. The correct return is to increment the address on the operand stack by one (1 FROM) and reference EMULATE with the currently emulated operation code and this address as operands.
2. On entry to the procedure, further emulation of the operation code is disabled so that the operation code may be used within the procedure without causing emulation. Emulation of an operation code is re-enabled by the reference to EMULATE which returns control from the emulation procedure.
3. Note that care must be taken within the emulation text from the standpoint of reuse of local variables in procedures which contain the operation code being emulated and which are referenced in the emulation code. For example, / is used in the OPINTD procedure. If / is being emulated and the emulation procedure uses OPINTD then such use will overwrite current values of local variables in OPINTD so that OPINTD will not continue normally after the emulated /. If emulation is being done using a previously undefined operation code problems of this kind will not arise.

   A sequence for emulation of the ENTRY primitive is:

```
FN EFT: ENTRY, OUTST('ENTRY executed at: '),
        OPINTD(DUP), OPNL, +1, EMULATE(34, <=>),
NOW EMULATE(34, @EFT) ! .
```

After the EMULATE operator has been executed, any execution of the ENTRY operator will cause the function EFT to be obeyed instead. Thus, the top item on the operand stack will be printed each time an ENTRY is executed, except that any ENTRY which is executed after entry into EFT and before the EMULATE instruction which causes return from EFT will be executed normally.

A more informative version of the above function would be:

```
FN EFT: ENTRY, OUTST('ENTRY into: '),
        LOOKD(DUP), OPNL, +1, EMULATE(34, <=>),
NOW EMULATE(34, @EFT) ! .
```

This sequence would display the name (if any) of the procedure being entered at each occurrence of an ENTRY primitive.

6.6.3 Virtual Machine Stops (STOP, ESTOP)

There are two operators which cause the Virtual Machine to stop instruction execution, one for normal stops (STOP) and one for abnormal or error stops (ESTOP). If MINT is implemented directly on a hardware system, STOP should terminate in such a way that a new MINT program may conveniently be loaded and run. ESTOP should terminate in a manner such that error data, such as the stack contents, may be displayed. Under a MINT implementation on an operating system, STOP should cause a normal return to the operating system. ESTOP should provide diagnostic information before taking an abnormal return to the system.

6.6.4 Host Environment Interface (EXR)

This operator, EXR, provides the means of communicating character string data to the host operating environment. The operator is necessarily host system dependent. The content and number of parameters depend on host system requirements.

6.6.5 Saving VSTORE (CDUMP)

The CDUMP directive is intended to provide a convenient means of saving the current contents of VSTORE so that computation may be continued at a later time. The restore of the saved VSTORE may be requested as a part of loading and initializing the MINT Virtual Machine. This operator may be implemented as an alternative to writing VSTORE in portable format, and then restoring it. In many implementations a direct save of VSTORE will be more efficient than using portable format. However, the direct save is not portable. The CDUMP operator references the PDUMP function to perform the writing of VSTORE in PDUMP format. CDUMP expects two arguments:

```
CDUMP 'filename' address
```

The filename is the name of the file in which to save VSTORE and address

is the address to set as the start address when the PDUMP is reloaded. Thus:

```
CDUMP 'MCOMP.PDM' 32768
```

will write out the system so that the compiler is initialized when the PDUMP is reloaded. This is the directive used to create the PDUMP form of the compiler.

The CDUMP operator may also be implemented to save the current VSTORE contents in portable format. In this case only a single VSTORE load mechanism is required. The choice of implementation techniques should be determined for each implementation depending on host-system characteristics.

6.6.6 Obtaining the Current Time (TIME)

The TIME operator obtains on the stack the current time in seconds. The exact form of the result of this operator may be implementation dependent.

6.7 Problems

6.7.1 Problem 1

Write text which will store the number 1 in the first location of an array, 2 in the second, etc., until values have been stored in the first 100 locations.

6.7.2 Problem 2

Write text to store the address of each element of an array into that element for an array of length 20.

6.7.3 Problem 3

Write text which determines how many of the elements of an array (of length 40) have a value which is greater than 10 and less than or equal to 45.

6.7.4 Problem 4

Write text which locates the first element in an array (of length 50) which has a value greater than 40.

6.7.5 Problem 5

Write text to sort the elements of an array of length 50.

6.7.6 Problem 6

Apply LIST LOCS LCODE to some of the previous problems to determine the apparent correctness and efficiency of the generated object text.

6.7.7 Problem 7

Write a procedure which moves the contents of table a to table b. The procedure is referenced as:

MOVE(length, address-of-table-a, address-of-table-b) .

6.7.8 Problem 8

Write a procedure which links a buffer into a chain of buffers. The procedure is to be referenced as:

LINK(address-of-control-block, address-of-buffer) .

The control block consists of two words. The first word points to the first linked buffer, or is zero if no buffers are linked. The second word points to the last linked buffer, or is zero. The new buffer is to be linked to the end of the chain. The first word of each buffer is to be used for linking the buffers.

6.7.9 Problem 9

Write a procedure which unlinks and returns the address of the first buffer in the buffer chain used in Problem 8. A value of zero is to be returned if no buffers are linked. The procedure is to be referenced by:

DELINK(address-of-chain-control-block) .

6.7.10 Problem 10

Write suitable procedures to trap instructions and display the address of each instruction, the instruction contents, and the current top-of-stack value of the operand stack.

7. Functions

7.1 Introduction

A function is a procedure which may be executed from anywhere within other text and which normally returns control to the point of reference when it is completed. In MINT there are two types of functions:

- Identified functions
- Anonymous functions.

7.2 Identified Functions

An identified function is a function which may be referenced by an identifier which has been introduced as class function (FN). Example function introductions are:

```
FN function
FN testit.
```

Within a procedure body the first operation must always be an ENTRY operation, i.e.

```
FN x: ENTRY .
```

Use of the EXIT operator at any point in the function returns control to the point following the function reference. A simple function would thus take the form:

```
FN xxx: ENTRY,
    text, ... ,
    EXIT .
```

7.2.1 Passing Parameters to Functions

Functions, since they are sequences of text which may be referenced from many places, are most useful when parameters can be passed to them and

they in turn can return results to the referencing text. Parameters are passed to functions by obtaining them on the stack and results are returned by leaving them on the stack before performing an EXIT. By the nature of the Virtual Machine any kind or combination of kinds of objects may be passed as parameters including the addresses of other functions. It is thus possible to implement such things as logical functions which return a Boolean result, or function obtaining functions which return as a result the address of some other function to be obeyed.

7.2.2 Referencing a Function

A reference to a function is generated simply by the occurrence of the name of an identifier which has been introduced as a function. Thus,

```
func
```

would compile the appropriate object text to reference the function func. Any parameters to be passed must have been obtained on the stack. For example,

```
a b 3 func
```

would obtain the values of a and b and the constant 3 on the stack (in that order) and then reference the function func. It is usual to express this more clearly by writing

```
func (a, b, 3)
```

with the parameters enclosed in parentheses following the function reference. The operation of this construction was explained in detail in the Section on the use of comma and parentheses (Section 4.9).

An example of a function to sum two numbers is as follows:

```
FN sum:ENTRY,                    . line 1
   +                             . line 2
EXIT                             . line 3
```

An example of a reference to this function is

```
sum (a, b) -> @c                 . line 4
```

Considering line 4 first, the function sum is referenced with two parameters which are the values of the variables a and b. The function sum is expected to obtain a single result which is stored into the variable c on return from the function. In line 1 the identifier sum is introduced as a function and set

(See Section 2.3.4). The first operation performed is the ENTRY operation. In line 2 the plus (+) operation is performed which sums the top two items on the stack and leaves the result as the top item. In line 3 the EXIT operation returns to the point where the function was referenced with the single result on the stack.

It is very important to remember that objects are popped from the stack in the reverse order from which they were pushed on. Consider a function max which obtains on the stack the greater of two values passed to it as parameters:

```
VAR arg1:0                              . line 1
VAR arg2:0                              . line 2
FN max:ENTRY,                           . line 3
    -> @arg2,                           . line 4
    -> @arg1,                           . line 5
    arg1 GT arg2 CHOOSE (arg1, arg2)    . line 6
EXIT                                    . line 7
```

This function might be referenced thus:

```
max (val1, val2) -> @biggest
```

In lines 1 and 2 two variables are introduced as temporary storage for the passed parameters. Line 3 is the function entry point. In line 4 the second parameter (val2, for the reference above) is taken from the stack. This is because val2 is the last parameter obtained and is therefore the top item on the stack.

7.2.3 Passing Parameters

Passing of parameters from a referenced function to another function is easily achieved by simply leaving the parameters on the stack, as in:

```
func1 (a, b)                            . line 1
func1:ENTRY                             . line 2
func2 (c)                               . line 3
EXIT                                    . line 4
```

In line 1 func1 is referenced with parameters a and b. In line 3 a second function is referenced with a single parameter c. Note that at this time there are three parameters on the stack, namely a, b and c. Thus, the two parameters a and b have been transparently passed from func1 to func2. For purposes of readability it is customary to write line 3 as

```
func2 (,, c)
```

the two commas being present simply to show that two parameters which were already on the stack are being passed to func2, in addition to the parameter c.

7.2.4 Complex Parameters

Parameters may be any form of object-obtaining expression. Consider a function called fill which presets table entries and is referenced with three parameters:

- The length of the table,
- The address of the table,
- The value to be set in each entry.

A reference to this function might be written as

```
fill ( &(@tabend ADIFF @table), @table, 2∗max(a, b))
```

where the first parameter is the evaluated constant &(@tabend ADIFF @table) (See Section 2.5.4.), the second parameter is a simple address constant, and the third parameter is an arithmetic expression which itself includes a reference to the function max with parameters a and b. All three parameters are fully evaluated before the function fill is referenced.

7.2.5 Functions Used to Define Record Structures

A record is a series of related data items, any one of which may be manipulated given the structure of the record and given the current address of the start of the record. Frequently, record structures involve pointers which allow substructures. The use of functions is a particularly effective means of expressing the accessing for such structures. For example, consider the record structure given in Figure 7-1. Assuming that the variable rpoint contains an address pointer to the start of the record, then the items could be referenced as:

```
VAL(rpoint),
VAL(2 FROM rpoint),
VAL(VAL(1 FROM rpoint),
VAL(1 FROM VAL(1 FROM rpoint)) .
```

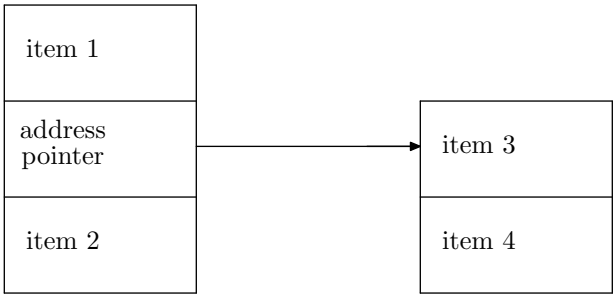


Figure 7-1. Record Structure

Such referencing is awkward and prone to errors which may not become immediately evident. Functions may be used to define such referencing structures in such a way that, within the referencing program, the references to each data item appear as normal variable references. For the above example, this is done as follows:

```
FN @item1: ENTRY rpoint EXIT
FN @item2: ENTRY 2 FROM rpoint EXIT
FN @srec: ENTRY 1 FROM rpoint EXIT
FN @item3: ENTRY VAL(@srec) EXIT
FN @item4: ENTRY 1 FROM @item3 EXIT
FN item1: ENTRY VAL(@item1) EXIT
FN item2: ENTRY VAL(@item2) EXIT
FN item3: ENTRY VAL(@item3) EXIT
FN item4: ENTRY VAL(@item4) EXIT .
```

Using the above functions, data items may be referenced or stored into as though the items were simple variables, as:

```
123 -> @item3,
item2 -> @item4 .
```

This technique has the additional advantage of providing program independence of the structure of the record. If a change is made to the record structure only the function definitions need to be changed.

7.2.6 Recursion

All MINT procedures can be used recursively since all return addresses are pushed onto the procedure linkage (link) stack. Recursion can be sup-

ported to any level. Note however that recursive use of variables is not automatically provided, and must be implemented by the programmer when required. MINT facilities for software controlled lists, and free-space management (See Chapter 9) make this a simple task.

7.3 Anonymous Functions

In cases in which a function is being passed as a parameter or otherwise used indirectly, it is not necessary to introduce a function identifier and write the function separately. Instead, the function may be written in-line and be unidentified. This facility is provided by square brackets, [and]. For example,

```
[0 ->@a, 0 ->@b]
```

is an anonymous function. When the compiler encounters such an anonymous function it generates the object text along with ENTRY and EXIT operations and sets up object text to obtain the address of the anonymous function on the stack. As an example of the use of an anonymous function, consider a function called calc which requires as parameters two values, and the address of a function which is to be referenced during execution of the calc sequence. The calc function might be referenced thus:

```
calc (a+b, 100, [0 ->@pqr, max(a, b)]) .
```

The third parameter is the address of the anonymous function which sets pqr to zero and obtains the result of max(a, b). Another example is the compiler's string input function INSTRING, which is passed the address of a pointer, and the address of a function which determines the end of the string. INSTRING may be referenced as:

```
INSTRING(@pointer, [NOCHS LT 20]) .
```

The anonymous function tests a compiler variable (NOCHS - the number of characters copied) and causes the INSTRING function to continue building the string while the number of characters is less than 20. The INSTRING function is described in Section 10.5.4.

7.3.1 The DO Operator

The DO operator is used to reference a procedure whose address has been obtained on the stack, or to reference a primitive whose operation code value has been obtained on the stack. The DO operator is useful, for in-

stance, if a variable is used to contain the address of a procedure or if a procedure address is passed as a parameter. This makes it convenient to write text which will execute different procedures, or primitives, depending on the current values of variables or passed parameters. The DO operator uses the value of the object on the stack to distinguish between its use for procedures and primitives. If the value is equal to or less than 80, DO executes the primitive. If the value is greater than 80 it makes a procedure reference. No check is made to ensure that the value is a valid operation code or that a valid procedure starts at the given address. The DO operator is the construct which is used to reference anonymous functions. Its form is:

```
DO address-obtaining-expression,
or numeric code of primitive,
```

where the address obtained is that of a function or directive. The DO operator can also be used to reference an identified function although the sequence is less efficient than direct reference. Thus,

```
DO @func
```

is directly equivalent to

```
func
```

since in the first case the address of the function func is obtained and the function referenced by means of the DO, and in the second case the function reference is generated due to the occurrence of the function identifier.

The use of the DO operator may be illustrated by reference to the calc example in the previous Section. The manner in which the described presetting facility is effected is as follows:

```
VAR presets:0
VAR arg1:0
VAR arg2:0
calc:ENTRY
    -> @presets,
    -> @arg2,
    -> @arg1,
    DO presets,
        :
        etc.
```

In this example the three parameters are stored on entry to the function. To reference the passed function the DO operator is applied to the value

of the variable presets, this value being the address of the passed function. Note that it is not strictly necessary to store the passed parameters. A similar function could have been written as

```
calc:ENTRY
    DO,
        :
        etc.
```

since the DO is applied to the top object (last parameter) on the stack. It is not necessary that the address of a passed function be that of an anonymous one, any function address will do. Thus,

```
calc (10, 20, @max)
```

would apply the max function discussed in previous examples to the first two parameters. Note that if max had been written without the at-sign (@) then instead of the address of max being passed, the function would actually have been referenced prior to referencing calc. This would have produced unintended results.

7.3.2 Link Stack Manipulation

Two operators are available to allow direct operation on the link stack pointer. These operators allow control over the effective return level from a procedure.

7.3.2.1 Obtaining the Link Stack Pointer Value (GLKP)

The GLKP operator obtains the current value of the link stack pointer (LP). Note that this is the pointer (LP) value, not the contents of the top of the link stack. This pointer value is not logically an address in VSTORE. The only valid use of the pointer value is to restore it to the link stack pointer by means of the SLKP operator. GLKP requires no arguments.

7.3.2.2 Setting the Link Stack Pointer Value (SLKP)

The SLKP operator resets the link stack pointer (LP). SLKP requires one argument which is a value which was previously obtained by means of GLKP. The effect of the SLKP operator is to adjust the link stack pointer so that the next reference to EXIT will return control to the procedure which referenced the procedure within which the pointer value was obtained by a GLKP. This provides a means of making direct returns to higher levels

without execution of a sequence of EXIT's. It may also be used to force a return which bypasses some text which would otherwise be executed. Since SLKP, in effect, causes abnormal flow of logical control, it should be used only when absolutely required and then with considerable care.

An example use of GLKP and SLKP is the following:

```
VAR linksave:0
FN SUB1: ENTRY
    SLKP(linksave)
        :
        other text
        :
    EXIT
FN MAIN: ENTRY
    GLKP->@linksave,
    SUB1
        :
    OUTST('Return from SUB1.'), OPNL,
    EXIT .
```

In this example, if MAIN is referenced, the EXIT in SUB1 will cause a return to the point at which MAIN was referenced. The text following the reference to SUB1 in MAIN will not be executed.

7.4 Miscellaneous Compiler Functions

7.4.1 The MOVE Function

The MOVE function physically copies data from one area of storage to another. It is referenced as follows:

```
MOVE (item-count,@source,@destination)
```

where the first parameter is the number of storage units to be moved, and the second and third parameters are the addresses of the source and destination areas respectively.

7.4.2 The DECMP Function

This function provides a means of displaying the contents of VSTORE in VM(M) instruction format. The D$ directive references DECMP to display

storage. The form of a DECMP reference is:

DECMP(start-addr, end-addr)

where start-addr is the start address and end-addr is the end address for the display. The display format is:

nnnnn label kkkkk oper, iden,

where:

nnnnn - VSTORE address,
label - identifier whose value is nnnnn, if any,
kkkkk - contents of VSTORE in decimal,
oper - operator whose value is kkkkk,
iden - identifiers whose values are kkkkk, if any.

7.4.3 The DUMP Function

The DUMP function simply stores the top item on the stack into a variable called TEMP. Since many procedures use DUMP and TEMP, the value saved in TEMP must not be considered to be preserved across compiler procedure references. TEMP is for temporary storage of intermediate results.

7.4.4 The NULL Function

The NULL function is a function which does nothing. It is provided for use in various situations where a function reference is required or desirable, but no action is required. For instance, the NULL function may be set in a table of function addresses in order to reserve a position in the table. Subsequently, that position may have another function address set into it.

7.5 Summary of Compiler Functions

The following Table gives the names of all normally available compiler functions. A brief description of the function is given. Some of these functions will not be fully described until subsequent Chapters. Refer to the Index for the location of the complete description.

Table 7-1 Compiler Functions

Function	Description
ADVCH	advance the input pointer by one character
BINLOC	binary search routine
BLANKS	set input pointer to next non-blank character
BTINIT	initialize B-Tree
BTDEL	delete entire tree
BTINSRT	insert data item in tree
BTREM	remove data (all items for given key)
CHAR	obtain current input character
COMPILE	compile a string
DECMP	display VSTORE in VM(M) format
DETACHED	pop item from item list
DIGIT	test if current character on input is digit
DUMP	save top of stack
FNAME	create string from input
FORBTVAL	apply a function to each data item for a given key
FOREACH	apply function to each item in list
GETSTR	read a string from an external segment
ININT	convert current input to integer
INSTRING	build input string
IPAR	evaluate current input expression
JOIN	join item to list
LETTER	test if current input is a letter
MOVE	copy data in VSTORE
NEXTCH	obtain next input character
NEXTFREE	acquire free-space block
NULL	null function
OPFF	output form-feed character
OPINT	display integer
OPINTD	display magnitude of number
OPNL	output carriage-return character
OUTST	display string
POPPEDUP	obtain item from item list
POPUP	pop item from list
POPUPREC	restore a record from a list
PUSHD	push item into list
PUSHDREC	save a record in a list
READ	read line into compiler input buffer
READINP	reference READ and do listing action
SAVBLOCK	decrement BLOCK level and save identifiers
SETBLOCK	increment level and reintroduce identifiers
SETBSE	set output conversion base
SETOPP	set output format

7.6 Problems

7.6.1 Problem 1

Convert the text for problems 1 through 5 of Chapter 6 to function form with appropriate arguments so that the functions may be applied to arrays of arbitrary length.

7.6.2 Problem 2

Modify the sort function so that the function which determines the order of two elements is passed as a parameter. Write a function for use with the sort function which will sort an array into odd values first, then even values.

7.6.3 Problem 3

Define a record structure suitable for telephone book entries, and write appropriate accessing functions.

7.6.4 Problem 4

Write a function to return on the stack the result of dividing two supplied parameters. The function is to be referenced as DIV(a,b) and is to return the result of a/b.

8. Directives and Immediate Execution

8.1 Introduction

A directive is a function which is obeyed when the compiler encounters a reference to its identifier. Much of the compiler itself consists of directives such as : (colon), @ (at-sign), PAGE, RENAME, etc. A directive is written exactly as a function with ENTRY and EXIT operations, the latter returning control to the compiler when the directive is completed. The directive's identifier is introduced into class DIR. Thus,

```
DIR abc
```

introduces the identifier abc as a directive. A directive cannot be obeyed until the identifier is set. Any reference to an unset directive identifier will produce incorrect results.

8.2 Input Parameters for Directives (IPAR)

Since directives are obeyed immediately, it is usual that any parameters which may be needed will follow as part of the input stream. Such parameters can be retrieved from the input image, evaluated, and obtained on the stack by means of a reference to the compiler function, IPAR. To illustrate this the compiler directive RESERVE will be discussed. This directive is written in the compiler as:

```
DIR RESERVE:ENTRY
        IPAR FROM DLOC -> @DLOC
        EXIT .
```

An example reference to the directive is:

```
RESERVE 500 .
```

When the compiler encounters the identifier RESERVE and determines that it is a directive it immediately executes the procedure. The first operation in the RESERVE procedure is a reference to the IPAR function which

then references the compiler to compile and evaluate the next element in the input stream, in this case the constant 500. The result of this evaluation is left on the stack. The IPAR function then returns control to the RESERVE directive where the parameter is used to increment the variable DLOC. (DLOC is the compiler's data location counter.)

8.2.1 IPAR-expressions

Because of the way the IPAR mechanism operates, only one MINT element is compiled as an IPAR parameter unless the first element is an open parenthesis. Thus,

```
RESERVE 2*a+b
```

would result in an IPAR value of 2, as the first element is not an open parenthesis. If an open parenthesis is the first element the IPAR function drives the compiler until a matching closing parenthesis is encountered. For this reason IPAR parameters consisting of more than one element must be enclosed in parentheses. Thus, correct forms are:

```
RESERVE (2*a+b)
```

or

```
RESERVE (2*(a+b)) .
```

Note in the second example that IPAR driven compilation terminates at the second closing parenthesis, i.e. the one which matches with the first open parenthesis.

IPAR-expressions are not constrained to obtaining a single object on the stack. For example, the compiler directive D$, which references IPAR, requires both a start and end address for its operation, i.e.

```
D$ (@func, 25 FROM @func) .
```

In this case two objects are obtained as the result of execution of the IPAR-expression.

8.3 Referencing Directives as Functions

As described previously, a reference to an identifier which is a directive causes the directive to be obeyed immediately. There are occasions when it is required to compile a reference to a directive, i.e. to treat a directive exactly as a function. This may be achieved by preceding the directive

name with the directive REF (See Section 2.3.3.4). Thus,

```
REF RESERVE
```

has the effect of compiling a function reference to the directive RESERVE. The compiler frequently makes use of this facility when it wishes to apply the action of the comma, i.e.

```
REF , .
```

A directive which uses the IPAR mechanism should not be referenced as a function unless it is really intended to cause the compiler to compile and execute the next input expression.

8.4 Immediate Execution

Directives provide a means by which text may be executed at compile time. However, the directives concerned occupy storage area and dictionary space. A facility exists in MINT to compile and execute text, and subsequently discard it and any identifiers associated with it. This is effected by means of the NOW and ! (exclamation mark) directives using the form:

```
NOW source text ! .
```

When this facility is used all the MINT source text between the NOW and ! is compiled and then executed. The space used for the compilation of this text is then released. This construction may be used to initiate a MINT program, for example:

```
LAB begin:
        :
        MINT program
        :
NOW GO begin !
```

As another example, if the reserve directive had not been implemented, its effect could be achieved by, for example:

```
NOW 100 FROM DLOC -> @DLOC !
```

instead of RESERVE 100.

The NOW directive is very useful for analysis and setup purposes, enabling functions in a program to be unit tested very conveniently without using up program space. For example, to trace a function execution the

following could be used:

```
T$(@NEWFN, @FNEND, 0),          . line 1
NOW NEWFN(10, 20) !             . line 2
NOW NEWFN(MINUS 4, 0) !         . line 3
T$(0, 0, 0)                     . line 4
```

Line 1 initiates trace mode between the limits @NEWFN and @FNEND (assume FNEND is a label in NEWFN) for all instructions. Line 2 contains a reference to NEWFN which is compiled and executed due to the NOW...! sequence. Trace information will be printed, along with any output from NEWFN. Line 3 causes the compilation and execution of another test of NEWFN. Finally, line 4 turns trace mode off.

8.5 The Class Directive

The CLASS directive introduces new identifier classes. When the new identifier is set a three entry table should be constructed as follows:

```
CLASS newcl: syntax, gen, assign
```

where newcl is the new class name, syntax is a function to be obeyed when an identifier in this class is recognized, gen is a function to be obeyed when object text is to be generated, and assign is a function to be obeyed when assignment action is required.

8.6 Miscellaneous Directives

The following directives provide diagnostic and program analysis tools.

8.6.1 The Decompile Directive (D$)

Areas of storage may be *decompiled,* i.e. their contents printed in Virtual Machine language (VM(M)) format. The decompile directive is invoked by

```
D$ (start-addr, end-addr)
```

where the two parameters are the starting and ending addresses of the area to be decompiled.

8.6.2 The Instruction Trace Directive (T$)

Execution of VM(M) Virtual Machine instructions may be *traced*. If an instruction is being traced its address, instruction text, and the current operand stack values are printed when the instruction is executed. Printing may be requested for a specified range of instruction addresses. In addition, the tracing of only a specific instruction code may be requested. The T$ directive is used as follows:

```
T$(start-addr, end-addr, op-code) .
```

The three parameters are defined as follows: If start-addr and end-addr are zero, tracing is disabled, otherwise tracing is enabled for the range of instruction addresses provided by start-addr and end-addr. When the op-code value is zero all instructions are traced and displayed, otherwise only the instruction whose value is given by op-code is traced and displayed. The T$ directive uses the Virtual-Machine instruction trap operator, TRAP, as discussed in Section 6.6.1. When T$ is referenced it displays a heading of the following form:

```
Trace mode active

LVL  LABEL     VA       VAL  OP.CODE   STACK:0     :1
```

where:

LVL	is the current relative level of procedure call, i.e. the relative value of the link stack pointer,
LABEL	is the name of any label which is set to this address,
VA	is the current VSTORE address,
VAL	is the instruction operation code value,
OP.CODE	is the name of the primitive, and
STACK	gives the top (0) and next (1) items on the operand stack.

Following this header, a line is displayed for each traced instruction, giving the information described above. If the traced text displays any output this output will be intermixed with the trace output.

In the current system the T$ directive is maintained in an optional file. It may be excluded from a copy of the compiler in order to minimize space requirements.

8.6.3 Dictionary and Dictionary Item Information

MINT dictionaries determine the behavior of MINT programs. Therefore, it is often useful to know the currently defined names in the active dictionaries, along with information about the names which is contained in the dictionary records. The directives ?, LV$, LISTDICS, OBJ, CURDIC, and PREVDIC provide information about the current dictionaries and dictionary entries.

The directive ?, or its alias CURDIC, displays the name of the current dictionary and the dictionary identifiers. If the variable FULDIC is not zero the CLASS and location of each identifier is also displayed. If FULDIC is zero only the names are displayed. FULDIC has this same effect when the LV$ (below) directive is used.

The LV$ directive displays the identifiers in each of the dictionaries in the current list from ENVLIST through LASTDICP. It also lists the dictionary name and count of items for each dictionary. All currently available names are listed in the order in which they occur in the dictionary. Note that, due to the dictionary hashing technique (See Section 9.7), this produces a listing which is "approximately" in alphabetical order. However, after the first character of the name the order will generally not be lexically correct. The information which is listed is the VSTORE address where the name was set and the class to which the name belongs.

The directive LISTDICS lists the name and item count for each dictionary in the current list from ENVLIST through LASTDICP.

The directive OBJ <object name> lists the properties and body of any object whose CLASS is known within the compiler. Specifically, if the object is a CLASS the syntax, generative and setting actions are shown, if the object is a FN or DIR the object text is displayed, if the object is a MACRO the text is displayed and if the object is a DICT, the identifiers in that dictionary are displayed. The properties shown are the shunt priority and the status word. In relevant cases the interpretation of the status word is printed. For example, after status 256 "Identifier not set." is printed.

The directive PREVDIC displays the identifiers in the "previous" dictionary. On the first use of PREVDIC, after a use of CURDIC, PREVDIC will display the contents of the dictionary previous to the current active dictionary. If PREVDIC is referenced again, it will display the contents of the next previous dictionary. Further references provide displays of further dictionaries until the end of the dictionary list, at which point the cycle will repeat.

8.7 Summary of Compiler Directives

The following table lists all normally available compiler directives with a brief description. Refer to the index for detailed descriptions of the use of each directive. A display of all idenifiers in the compiler dictionary MAINDIC is easily obtained by: OBJ MAINDIC. The compiler directive UNLOCK INTDIC (this is the replacement for the old ICL$) makes available all internal compiler identifiers. These identifiers are not usually required. They are defined in the compiler text and are introduced in the dictionary INTDIC.

Table 8-1. Compiler Directives

Directive	Description
!	close body of immediately executable text
$	convert HEX constant
'	compile string constant
(	delimit precedence operation
)	delimit precedence operation
,	terminate shunt action
.	comments follow
\	dictionary reference
%	dictionary reference
(	determine evaluation order
)	determine evaluation order
:	set an identifier to a location counter
;13	C/R character. Ends current input line
?	display current dictionary contents
@	compile address constant
#	compile character constant
&	compile evaluated constant
BLOCK	open local identifier block
CDUMP	write VSTORE to a file
CURDIC	display current dictionary contents
D$	decompile an area of storage
DATA	set data state
DPRMPT	set prompt
ENDBLOCK	close local identifier block
EQV	set an identifier to a parameter
FORGET	remove identifier
ICL$	UNLOCK INTDIC
LCODE	enable listing of VM(M) object code
LIST	enable listing of source
LISTDICS	list names of all active dictionaries
LOCK	exclude dictionary from lookup path
LOCS	enable listing with location counters
LV$	display identifier information
MINUS	compile negative integer constant
NOLIST	disable listing
NOPRMPT	turn off prompt
NOW	open body of immediately executable text
OBJ	display body of identifier

OBREAK	close output file and redirect output
PAGE	skip to a new listing page
PREVDIC	display contents of previous dictionary
PRIORITY	set identifier priority
PROG	set program state
RCL$	LOCK INTDIC
RD-ONLY	make dictionary not modifiable
RECPOOL	show RECPOOL usage
REF	reference a directive as a function
RENAME	rename identifier
RESERVE	reserve data space
SI	redirect input to specified file
SO	redirect output to specified file
STOP!	exit the VM
T$	enable or disable instruction trace
TITLE	reference PAGE and print heading
UNLOCK	include dictionary in lookup path
VSTOREMAP	show VSTORE usage

8.8 Problems

8.8.1 Problem 1

Write a directive named HELP which prompts the user for the keywords DIR, FN, VAR, and MACRO and prints a short description in response to each keyword.

8.8.2 Problem 2

Use the D$ directive to display the object text of the RESERVE directive.

8.8.3 Problem 3

Write a directive which acts like D$ but requires only the starting address and displays object text until the next EXIT.

8.8.4 Problem 4

Why do the D$ and T$ directives not need to be referenced from within a NOW...! sequence?

8.8.5 Problem 5

Write directives similar to the LV$ directive which list dictionary records selected by class and by setting address range. This problem could be solved directly, or after study of the text of the LV$ directive which forms a part of the distributed machine-readable system.

9. Lists and Free-Space Management

9.1 Introduction

Within the MINT compiler a number of facilities exist to acquire and release areas of free storage, to push and pop data to and from stack-like lists and to manage such lists. This Chapter describes the functions that are an integral part of the compiler and may be freely used by the user as part of the run-time system.

9.2 Basic List Structure

The basic list structure which is supported by compiler functions is a *lifo* (last-in-first-out) forward linked structure. The first record in the list is addressed by a pointer. Each subsequent record is pointed to by the first word of the current record. Records of any length may be chained in the list. Figure 9-1 shows the structure of these lists. The last record in the list contains a pointer whose value is zero to indicate the end of the list.

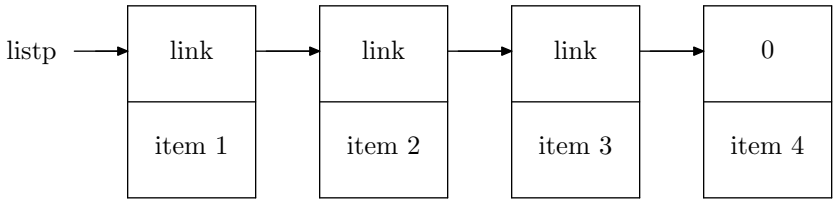


Figure 9-1 Basic List Structure

These records were pushed (See Section 9.5 for the item manipulation functions) onto the list in the order 4, 3, 2, 1. The first removed item would be item 1.

9.3 Adding to and Removing from a List

Two functions are provided which add a record to the top of a list and remove the top record from a list. These are respectively:

JOIN
DETACHED.

9.3.1 The JOIN Function

The JOIN function is referenced as follows:

```
JOIN (@block, @listp)
```

where the first parameter is the address of a record to be linked into the list, and the second parameter is the address of a variable to be used as the list pointer. The effect of referencing JOIN is that the address of the record is stored in the list pointer, thus making the record the top of the list. The original content of the list pointer is stored into the first word of the record, thus linking it to any previous blocks in the list.

9.3.2 The DETACHED Function

The DETACHED function performs the opposite action of JOIN and is referenced as follows:

```
DETACHED (@listp)
```

where the single parameter is the address of a list pointer. The effect of a reference to DETACHED is that the address of the first record on the list is obtained on the stack. The content of the first word of the detached record is stored into the list pointer. Thus,

```
DETACHED(@listp) -> @blokaddr
```

results in the address of the detached record being stored in the variable blokaddr. At the same time the list itself is adjusted by storing the contents of the first word of the detached record into listp, i.e. the previously second record in the list becomes the top one.

9.4 Free-Space Management

The compiler manages a data area termed *free-space*. This space is allocated at the high end of data-space. Free-space provides an area from which blocks

of storage of various lengths may be acquired as working space. When such blocks of storage are no longer needed they may be JOIN'ed to a list pointer called a free-space list and thereby made available for later use. A free-space list is a two word record containing:

- List pointer
- Storage block length.

Thus:

```
VAR fslist: 0,8
```

describes a free-space list (initially empty). The length of the blocks in the list is 8 storage units.

9.4.1 Acquiring Free-Space

Free-space blocks may be acquired by referencing the compiler function NEXTFREE as follows:

```
NEXTFREE (@fslist)
```

where the parameter is the address of a free-space list. The NEXTFREE function obtains on the stack the address of a block of storage of the length prescribed by the free-space list. It does this in the following manner. If the free-space list is not empty (i.e. list pointer non-zero) NEXTFREE references DETACHED to remove the top free block on the list. If, on the other hand, the list is empty (list pointer zero) the number of storage units required is taken from the free-space area and the address of the block thus obtained is passed back to the user. Note that in this case the free-space list pointer (fslist) is unaffected as the list itself is still empty. Thus, consider:

```
VAR bufaddr:0
VAR buflist:0, 100
NEXTFREE (@buflist) -> @bufaddr .
```

In this example the address of a 100 word block of storage is acquired from free-space and stored in the variable bufaddr. The free-space list is called buflist.

9.4.2 Releasing Free-Space

Free-space blocks are released by simply JOIN'ing them to a free-space list. Thus,

```
JOIN (bufaddr, @buflist)
```

(using the same variables as those described in the previous Section) performs the following action: The value of the variable bufaddr is the address of a 100 word block of storage. This block is JOIN'ed to the list pointer, buflist. A subsequent reference to the function NEXTFREE would obtain this buffer address. Note that the user is responsible for releasing the correct size of block to the appropriate free-space list.

9.5 Item Lists

The compiler itself makes considerable use of two word records called *item blocks*. These item blocks are linked into a number of internal lists and are used to store single word items (see Figure 9-1). Such internal lists are effectively software stacks. The functions described below are provided to support these stacks.

9.5.1 The PUSHD Function

The PUSHD function is used to store a single value into an item block and JOIN the item block onto a list. Its form of reference is:

PUSHD (item,@listp)

where the first parameter is any item (value, buffer address, etc.) and the second parameter is the address of a list pointer. PUSHD operates in the following manner. First, the function NEXTFREE is referenced to acquire a two word item block from free-space. Secondly the passed item is stored in the second word of the item block. Finally the function JOIN is referenced to link the item block to the specified list pointer.

The PUSHD function provides a means of obtaining an object on a software defined stack.

9.5.2 The POPUP Function

The POPUP function is referenced to discard the top item and its associated item block from an item list. The form of the POPUP function reference is:

POPUP (@listp)

where the single parameter is the address of a list pointer. The POPUP function references DETACHED to remove the top item block from the list and references the function JOIN to release the block to the internal

two word free-space list. No result is returned from the POPUP function. It therefore operates in a manner analogous to the LOSE operator.

9.5.3 The POPPEDUP Function

The POPPEDUP function is referenced to obtain on the stack the item contained in the first item block of an item list, and POPUP the item block. The form of the POPPEDUP reference is the same as that for POPUP. Thus,

```
VAR item:0
POPPEDUP (@listp) -> @item
```

obtains the top item from the item stack pointed to by listp and stores it in item. The POPPEDUP function is thus the converse of the PUSHD function. Note that the top item of a list may also be obtained without discarding it by the expression:

```
VAL (1 FROM listp)
```

since listp contains the address of the item block, and the VAL obtains the contents of the second word which is the item itself.

9.5.4 The FOREACH Function

The FOREACH function applies a passed function to each item in an item list. Its reference form is:

```
FOREACH (@listp,@function) .
```

It operates by obtaining the item from each entry in the list and referencing the passed function. The passed function is expected to remove the obtained item from the stack. For example, consider a list of items which are addresses of store locations whose contents must be adjusted by a factor rbias. Then

```
FOREACH (@listp, [DUMP, VAL(TEMP)-rbias->TEMP])
```

will, for each address in the list, reference the anonymous function [...]. The anonymous function references DUMP to put the address into the variable TEMP. The content of the location is adjusted by rbias and the result stored back into the location whose address is in TEMP. Note that TEMP is used as a pointer.

9.5.5 Deleting an Item List

The function CLEARLST(<@listp>) is available to perform a POPUP on each item in the list.

9.5.6 Operation on Characters of a String (FORCHS)

The FORCHS function provides a means of applying a procedure to each character in a string. It is referenced as follows:

FORCHS(<string address>, [function])

where [function] is a function that expects a character on the stack and returns its character result on the stack. FORCHS obtains a character from the string, references the function, and pushes the top item on the stack back into the character position of the obtained character. This operation is performed on each character in the string in order.

9.6 Record Lists

In many situations, instances of a record or table of pointers and data items are required to be saved, and at some later stage restored. The compiler provides functions to perform these operations. The functions involved may be compared with the PUSHD and POPPEDUP functions described above, but instead of saving and restoring single items they operate on contiguous blocks of store and save and restore the contents of the entire block of store. Such blocks, or records, are described by a record list pointer which consists of two items:

- List pointer
- Record length.

Thus, in the sequence:

```
VAR record:
VAR entry1:0,100
VAR entry2:abc def
VAR entry3:MINUS 10
VAR rlistp:0,&(@rlistp ADIFF @record)
```

the variable rlistp forms a pointer to such a record list. The first word is the list pointer (initially empty) and the second word is an evaluated constant whose value is the length of the record (5 in this example).

The list structure which is used by these functions is shown in Figure 9-2. The Figure shows the structure in its state after four PUSHDREC references (pushing the records in the order 4, 3, 2, 1) followed by two POPUPREC references. Initialization is automatically carried out by PUSHDREC (See below).

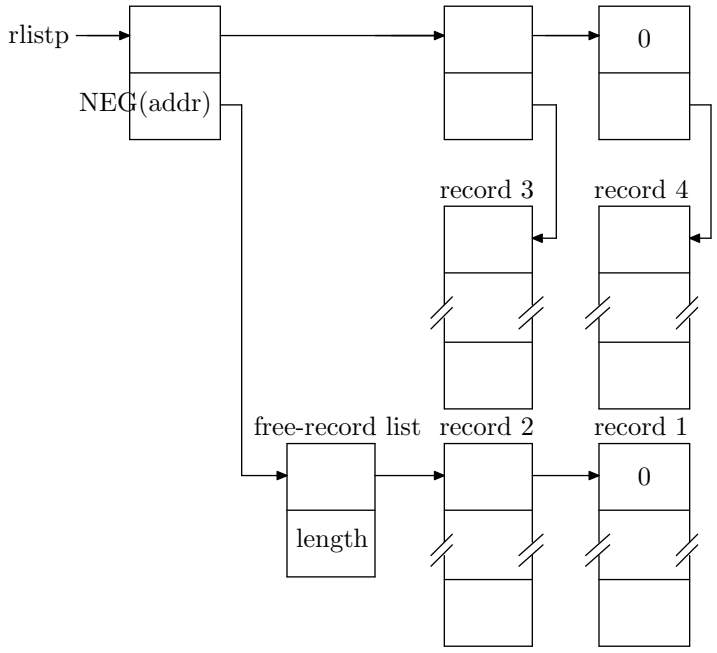


Figure 9-2 Record List Structure

9.6.1 Saving Records

Records are saved by referencing the PUSHDREC function as follows:

PUSHDREC (@record,@rlistp)

where the first parameter is the address of the record to be saved and the second parameter is the address of a list pointer record. PUSHDREC operates on two lists. After initialization, the list pointer record (rlistp) contains the addresses of these two lists. Initially, the list pointer record contains zero in the first entry and the intended record length in its second entry. As a part of initialization, PUSHDREC stores NEG of a list address

in the second entry. Thus, the sign of the value in the second entry is used to indicate if initialization is to be carried out. PUSHDREC carries out the following steps:

1. If the second entry of the list pointer record is positive NEXTFREE is referenced to obtain a two word item record. The length from the second entry in the initial list pointer record is moved to the second word of the item record, and NEG of the address of the item record is stored in the second word of the list pointer record. If the second entry in the list pointer record was already negative PUSHDREC proceeds to the next step below.
2. NEXTFREE is referenced with the address of the item record, in order to obtain a record of the required length. Initially, the item record will indicate an empty list (zero in the first entry) and space will be acquired from free-space.
3. The record to be saved is copied into the obtained record.
4. The address of the obtained record is stored in an item which is pushed onto an item list by a reference to PUSHD using the first entry of the list pointer record (rlistp) as the item list pointer.

9.6.2 Restoring Records

Records saved by the PUSHDREC function may be restored into any storage area of the appropriate size by means of the POPUPREC function:

```
POPUPREC (@record,@rlistp)
```

where the first parameter is the address where the record is to be written, and the second parameter is the address of a record list pointer. The POPUPREC function operates as follows. First the function POPPEDUP is referenced to obtain the address of the saved record. Second the record is copied into the user specified address. Finally the saved record is released to the free-record list.

9.7 Variable Length Records

The compiler provides for the management of a pool of variable-length records. This mechanism is used to manage the dictionary records which are described in the next Section. The space for these records is acquired from free-space in blocks as needed and space is recovered by recombining released records as possible.

9.7.1 Acquiring a Record

The function which obtains a variable-length record is:

```
GETREC (@rlength).
```

This function requires the record length as its parameter, and returns the address of the record.

9.7.2 Releasing a Record

The function which releases a variable-length record is:

```
RELREC (@record, @rlength).
```

This function releases the record at the address given as the first parameter. The second parameter must be the length of the record as specified in the GETREC reference. It is necessary for the correct operation of the record pool mechanism that only valid record addresses and lengths be used in RELREC references.

9.8 The Dictionary List Structure

Dictionaries use a list structure similar to that described in Section 9.2. The format of the records is shown in Figures 9-3 and 9-4. Each dictionary record consists of a link word, a MINT formatted string, and three words which contain the identifier address, class and level respectively. Note that the link word in the last valid dictionary record contains the address of the end of the dictionary (dicend). The record pointed to by dicend is not a valid dictionary record. The dictionary records are variable-length records. For an identifier of length N, the dictionary record length is:

```
5 + ((N+3)/4).
```

Figure 9-4 depicts the structure of a dictionary record.

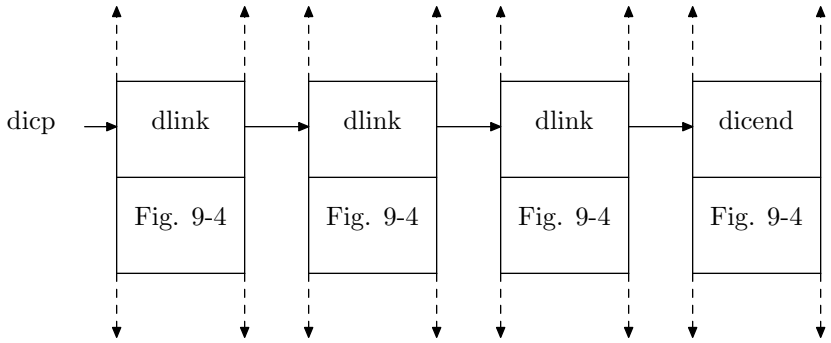


Figure 9-3 Dictionary List Structure

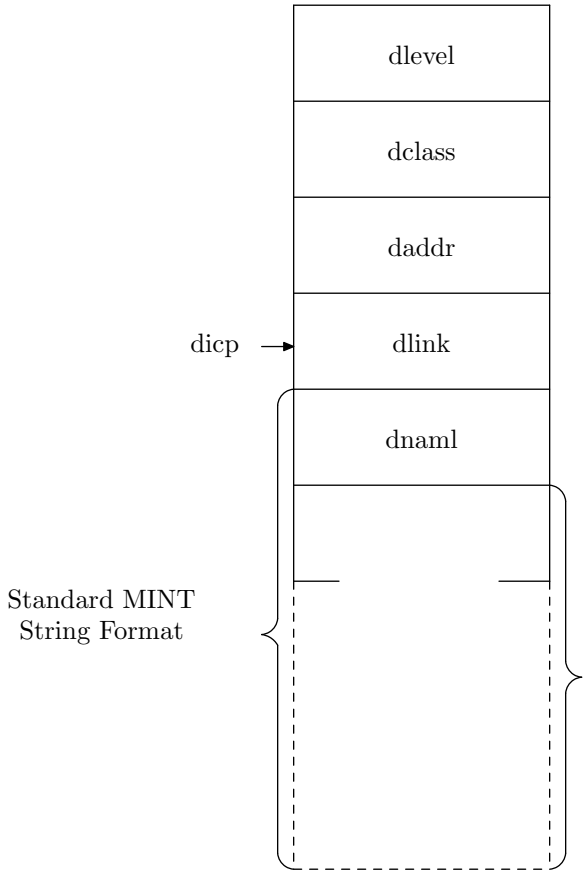


Figure 9-4 Dictionary Record

Each dictionary has 64 dictionary lists, all of which link to the dictionary end record. The selection of the correct dictionary list is based on a hash function which uses the low order 6 bits of the first character of the identifier name. This structure is simple, but provides quite short average identifier search paths as long as the total number of identifiers remains less than of the order of 1000. In addition, this provides an approximately lexical ordering of entries in the dictionary.

9.9 B-Tree Data Access

The purpose of these routines is to provide a B-tree access structure for storing and retrieving data items or records within MINT VSTORE. The routines provide the following functions: initialization, storage of data for a given key, deletion of data, and a function that permits operation on each data item stored under a given key. Nodes in the tree are filled as new keys are stored, and split when they become full. At present, there is no provision for compacting the tree structure or deleting keys. If all the data items for a given key are deleted the key sequence will remain, with an empty item list at the end of it. Leaf nodes are split without increasing the tree depth and all non-terminal nodes are split by introducing a new node that increases the depth of its branch by one.

The node search function is supplied as an argument when a tree is initialized. Nodes contain single-word entries for keys. However, the search function may interpret these entries as addresses and thus may perform the comparison on any user-defined structure.

9.9.1 B-tree Functions

The functions described below provide for creation and manipulation of the B-tree data structures and access to the data.

9.9.1.1 BTINIT

This function creates the initial tree structure. It requires four arguments: the address of a pointer to the tree (for future reference), the address of the key compare function, and lower and upper limits for the expected key values. Thus, a reference is:

BTINIT(<lower limit>, <upper limit>,[key compare function],
<tree pointer>)

The *key compare function* is a function which locates the correct entry in a node, given the key. The user is required to provide this function in order to permit the B-tree mechanism to be used with key structures of the user's choice. If the key is a one-word item it may be stored directly in the node. Otherwise, the entries in the nodes should be addresses of key objects (such as variable length strings). If the key entry in the node is an address then the key values supplied for all other B-tree functions must be addresses of compatible objects. The key compare function must be written to expect three arguments and to return one argument. The three input arguments are:

1. *key*. This is either a key value or the address of a key structure. It must have the form used in all the other function references.
2. *@first entry*. This is the address of the first entry in the table to be searched.
3. *number of entries*. Number of entries to search.

Each entry in the table pointed to by the second argument is two words long. The key address or value is in the first word and a pointer is in the second word. Therefore, the search routine should increment by two as it compares table entries.

The returned argument is the address of the second word (pointer) of the entry before the entry containing the key greater than or equal to the argument key. If no key is found that is greater than or equal then the address of the second word of the last entry is returned.

The address of the tree pointer is required as an argument in all of the functions so that multiple trees may be in use at any time.

9.9.1.2 BTDEL

This function deletes the entire tree and returns all node and data item records to the record pool. Its reference is:

BTDEL(<tree pointer>)

9.9.1.3 BTINSRT

This function inserts a data item under the key provided. Its reference is:

BTINSRT(<data>, <key>, <tree pointer>)

9.9.1.4 BTREM

This function removes all data items stored under the given key. Its reference is:

BTREM(<data>, <key>, <tree pointer>)

9.9.1.5 FORBTVAL

This function applies the provided function to each data item stored under the given key. Its reference is:

FORBTVAL(<key>, [<function reference>], <tree pointer>)

9.9.2 Data Structures

There are two data structures: non-terminal nodes and leaf nodes. They are shown below:

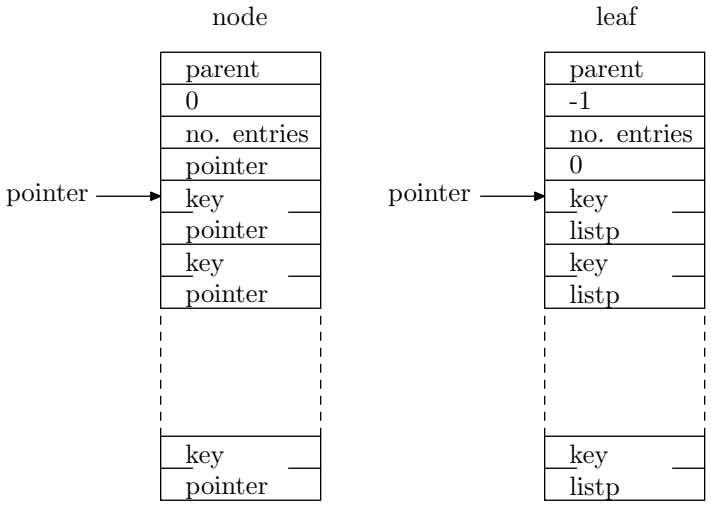


Figure 9-5 Node and Leaf Data Structures

9.10 Problems

9.10.1 Problem 1

Define an item list composed of the items 5, 3, 1, 7, 9, and 4.

9.10.2 Problem 2

Write a function to insert an item in the list defined in Problem 1 after a given item value.

9.10.3 Problem 3

Write a function to sort the items in the list defined in Problem 1.

9.10.4 Problem 4

Define a record to contain names and addresses. Write a function which will insert such records in a list in alphabetical order by the last (family) name.

9.10.5 Problem 5

Write a function to delete a name from the list defined in Problem 4.

9.10.6 Problem 6

Write a function to print all records in the list defined in Problem 4 which contain a given first name.

9.10.7 Problem 7

Write functions which perform according to the specifications of DUMP, MOVE, JOIN, DETACHED, PUSHD, and POPUP. After these functions are written, compare the text with the functions as given in the compiler text.

10. The External and String Operators

10.1 Introduction

The MINT compiler contains a set of procedures and primitives for handling strings, individual characters within strings, and for communicating into and out of VSTORE. This Chapter describes these procedures and primitives.

MINT character and string handling depends on characteristics of the ISO/ANSI character set. Correct operation will not result if another character set is used. The full ANSI set is supported except that NUL (value 0) is treated as the end-of-string indicator. (However, a NUL character may be transmitted by use of the value 0400, since the end-of-string test is a test zero on all eight bits in the character field.) Variations in the graphic representations of ISO/ANSI characters, such as national definitions different from the U.S. definition, will not affect MINT operation, but may affect readability. This will be especially true if such characters as [,], @, <, or > have alternate graphics.

10.2 MINT String Format

MINT strings consist of a one word character count followed by a series of ANSI characters stored 4 characters to a 32 bit MINT word. The first character in a word normally occupies the low order (least significant) 8 bit byte, and the characters follow in increasing byte order as shown in Figure 10-1. The high order bit of each character should normally be zero.

The order of the characters in a word is possibly implementation dependent. Some implementations may have the order within a word reversed from the usual low order byte first sequence. This reversal may be required for efficiency. (The PDP11 was an example of a machine in which reverse order is required and others still exist.) If the standard MINT facilities are used, the character order is not normally significant. The main exception to this rule is the creation of portable format output on one system which is to be loaded into another system which uses a different character order. A

procedure is available for carrying out the character reversal transformation which is required in this case. This is discussed more fully in Chapter 14.

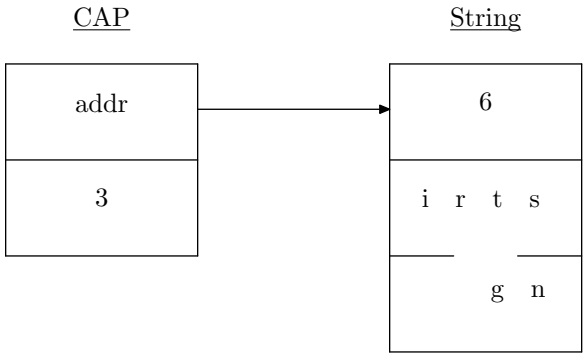


Figure 10-1 MINT String Format

A MINT formatted *string* is defined as shown in Figure 10-1. MINT formatted strings are referenced by means of a Character Address Pair (CAP). A CAP consists of two storage units (See Figure 10-1) the first of which contains the address of the first word of the string, and the second contains a character index. The index is an integer number, the first character in the string being referenced by an index value of 0. Thus in Figure 10-1 the CAP references the character "i".

10.3 Initialization of External Segments

It is necessary to initialize access to any external segment except segment zero. This is done by means of the OPENF primitive.

10.3.1 The OPENF Primitive

The OPENF primitive is a Virtual-Machine operator which requires two parameters. It opens an external segment for input, output, or both. The first parameter determines the type of initialization to be performed. Table 10-1 shows the standard type definitions. Specific Virtual Machine implementations may extend the type definitions to provide for environment dependent requirements. The standard types given in Table 10-1 should not be changed.

Table 10-1. OPENF Operation Types

type	meaning
1	Sequential input
2	Sequential output
3	Direct access input/output
4	Magnetic tape input/output
5	Communications input/output

The second parameter is the address of a MINT formatted string which is the name of the segment to be opened. The content of the string is implementation dependent, as is the number of segments which may be open concurrently.

OPENF returns one parameter which is the index assigned to the external segment. This number is a positive integer greater than zero. However, the value is otherwise implementation dependent and should not be considered to be fixed for any given segment name. The index is used to reference an opened segment for any subsequent operations. Thus a typical OPENF reference is:

```
OPENF(1, 'external name') ->@index1 .
```

This sequence has the effect of initializing the external segment whose name is "external name" for sequential input, and storing the segment index in the variable index1.

If type index 3 is used, SEGIO should be used to reference the external segment. SEGIO provides record-structured input and output. It is fully described in Section 13.5.12.5.

10.4 Input Facilities

Input consists of characters, which may be composed into strings. The Virtual Machine obtains characters, using the INCH primitive. A function, GETSTR, is available which uses INCH to compose strings. Each string may contain up to a specified number of characters, and is terminated by the carriage-return character. Input is obtained from *external segments*. These logical segments may be files, devices, or other external facilities. Each active (open) external segment has a unique index. The indices are assigned by the VM(M) Virtual Machine. Segment index zero (0) is always associated with the *primary* I/O device.

10.4.1 The INCH Primitive

The INCH primitive is a Virtual Machine operator which reads a character from an external segment. It requires one argument which is the input segment index. It returns a single argument which is the next character from the segment. If the end of the segment has been reached INCH will return the hex value 0x100 to indicate the end-of-segment condition. The form of an INCH reference is:

INCH(index1) .

This causes the Virtual Machine to obtain the next character from the segment whose index is index1, or obtain the value 0x100 in case the end-of-segment has been reached. The Virtual Machine uses the C/R character to indicate end-of-line. Thus, if a segment contained the two lines:

abcd
xyz

the sequence that would result from successive INCH references would be:

a b c d C/R x y z C/R 0x100 0x100 ...

10.4.2 The GETCH Primitive

The GETCH primitive obtains on the stack a single character from a MINT formatted string. It requires a single parameter, namely the address of a CAP referencing the string. The form of the GETCH reference is:

GETCH(@CAP) .

If the CAP in this example were that illustrated in Figure 10-1 then the character "i" would be obtained on the stack. The character index is not incremented.

When a GETCH operator is executed and the CAP parameter points to a character beyond the range of the string, a zero (0) value will be returned. Thus, a zero character value should not be contained within a string.

10.4.3 The GETSTR Function

The GETSTR function uses the INCH primitive to read characters from an external segment and compose them into a MINT string. It requires three

parameters which are the segment index, the maximum number of words to be used in the buffer, and the address of the buffer area in which the string is to be composed. The form of the GETSTR reference is:

GETSTR(index1,mxlength,@inbuf) .

GETSTR references INCH to obtain characters. These characters are composed into a MINT formatted string in inbuf. Input characters are read until an end-of-image condition (usually a C/R character), or until the buffer length limit (mxlength) is reached. The last character in a string formed by GETSTR is always a carriage-return (C/R). Since the first word of the input buffer contains the string length and a C/R is appended at the end, the number of words required for an input string of length N characters (not including the C/R) is:

L = 2+(N/4) .

The end-of-file condition is indicated by means of the character count contained in the first word of the input buffer. A character count of zero is treated as the end-of-file condition. Note that an *empty* string will not have a character count of zero since the C/R character must be present, and is always included in the character count.

10.5 Compiler Input Facilities

The compiler uses standard input facilities to read its input. User text may, of course, reference the compiler input mechanisms.

The compiler input CAP consists of the variable CURCHS, containing the address of the current input image, and the variable CURCOL, which is the character index. All the input procedures described in the following Section operate on CURCHS, thus addressing the compiler input image. SIUNIT contains the compiler's current external segment index for compiler input. SIUNIT is an item in the compiler record whose list is pointed to by the variable INPST. This record contains SIUNIT and LINENO, the current input line number.

The MINT input routine does not treat the end-of-line as significant. Lines are logically concatenated so that expressions of any length may be composed. See Section 2.5.5 for handling of multi-line literal strings.

10.5.1 The SI Directive

All compiler input procedures operate on the external segment whose index

is contained in the variable SIUNIT. The SI directive may be used to direct the compiler's input stream to another external segment. The form of the SI reference is:

SI character-string .

The character-string is a string, terminated by a blank character or carriage-return, which is transferred to the OPENF primitive to specify the external segment name. The record containing SIUNIT and LINENO is pushed onto the record list whose pointer is INPST, SIUNIT is set to the value returned by OPENF, and LINENO is set to zero. When an end-of-file is encountered the current input is closed and a POPUPREC(@SIUNIT, @INPST) is performed. This restores the previous input segment and recovers the line number within that segment. When an end-of-file for segment index zero is detected the MINT system is terminated.

10.5.2 The READ function

The READ function reads a new input string into the compiler's input buffer. The buffer is identified by the CAP CURCHS. The variable CURCOL contains the current character pointer (i.e. it is the second word of the CAP). If CURCHS does not point to a buffer, READ will assign a buffer. The variable SIUNIT is used to obtain the input segment index. On each READ the variable CURCOL is set to point to the first character of the string. The READ function requires no parameters.

10.5.3 The READINP function

The READINP function uses the READ function to read a new image into the compiler's input buffer. In addition, it updates the compiler's output listing line number (LINENO) and then applies any listing options (See Chapter 3).

10.5.4 The INSTRING Function

The INSTRING function constructs a MINT formatted string from the current compiler input string. It requires two parameters:

1. Address of destination pointer
2. Address of delimiting function.

The destination pointer contains the address where the string is to be built. On return from INSTRING this pointer contains the address of the first

word beyond the composed string. The delimiting function is applied to each character copied, and must obtain a Boolean object. String composition continues until a false value is obtained by this function. Thus, for example

```
DIR ':ENTRY
        INSTRING(@DLOC, [CHAR NE #'])
                :
                :
```

illustrates how the quote directive composes a string. The destination pointer is the compiler's data location counter, DLOC. The delimiting function tests if the current character (CHAR, see below) is a quote. If it is, then the string composition terminates. During string composition the INSTRING function increments a counter, NOCHS. This counter contains the number of characters copied into the destination string. NOCHS may be tested in the delimiting function, or referenced subsequently.

10.5.5 The BLANKS Function

The BLANKS function skips blanks in the input image. It requires no parameters. CURCOL is incremented to point to the next non-blank character.

10.5.6 The ININT Function

The ININT function generates an integer number from a string of digits in the input image and obtains the integer on the stack. The CURCHS CAP must be positioned on a digit when ININT is referenced. Thus, if the current input contains blanks followed by one or more digits,

```
BLANKS
ININT -> @value
```

will store the resulting integer in value. The maximum value correctly accepted by ININT is $2^{32} - 2$. If the current character is not a digit, zero is returned and CURCOL is not incremented.

10.5.7 The INHEX Function

A function INHEX is included for reading numbers in standard HEX format. It operates just like ININT, but it expects the digits 0-9,A-F.

In addition, the directive $ reads the immediately following constant using HEX format. Thus, VAR XX:$FF would assign the value 255 to the variable XX.

10.5.8 The CHAR Function

The function CHAR obtains on the stack the current character from the compiler's input image. It requires no parameters. If CHAR is referenced when the character index, CURCOL, is beyond the end of the string, automatic end-of-string action is performed. This is generally a reference to the function READINP. Note that CHAR does not increment CURCOL, and does not act on the semicolon to translate special sequences.

10.5.9 The ADVCH Function

The ADVCH function advances the character pointer, CURCOL. It requires no parameters. An example of its use is the BLANKS function:

```
FN BLANKS:ENTRY
        WHILE CHAR EQ # START
                ADVCH
        REPEAT
EXIT .
```

10.5.10 The NEXTCH Function

The NEXTCH function obtains the current character from the input image, and advances the character pointer, CURCOL. If the character so obtained is a semicolon (the *escape* character), special processing is applied, otherwise the character is returned. The semicolon indicates that subsequent characters are to be translated to a single unique character. Thus, for example, ;CR is translated to the C/R character (octal value 015). The sequences which have defined translations are contained in a table named ESCTAB. The NEXTCH function attempts translation whenever a semicolon is detected. If the translation process is successful (i.e. a match is found in ESCTAB), the corresponding single character is returned and CURCOL is advanced beyond the matched sequence. If it is not successful, the character beyond the semicolon is returned and CURCOL is positioned beyond it. Thus,

```
;;              will return a single ;
```

;FF	will return a form-feed (014)
;CR	will return a carriage-return (015)
;S	will return a space character (040)
;'	will return a single quote character (054)

The NEXTCH function requires no parameters.

10.5.11 The DIGIT Function

The DIGIT function references CHAR and determines if the character is a digit. It returns a Boolean true if CHAR is a digit and false otherwise. The DIGIT function definition is:

```
FN DIGIT: ENTRY
        CHAR GE #0 AND CHAR LE #9
        EXIT .
```

10.5.12 The LETTER Function

The LETTER function references CHAR and determines if the character is alphabetic. It returns a Boolean result accordingly. The LETTER function definition is:

```
FN LETTER: ENTRY
        CHAR GE #A AND CHAR LE #Z
        OR (CHAR GE #a AND CHAR LE #z)
        EXIT .
```

10.5.13 The FNAME Function

The purpose of the FNAME function is to create a string from the current compiler input source into a temporary buffer. The address of the string is returned on the stack. FNAME requires no arguments. FNAME references BLANKS and then references INSTRING to read the string, with the termination condition set to a blank or C/R. The temporary storage which is used for the string starts at the current DLOC value. The value of DLOC is not changed.

10.6 Output Facilities

Output characters may be directed to the primary I/O device (segment

index zero), or to other external segments in a manner analogous to that described for input.

10.6.1 The OPCH Primitive

The OPCH primitive transmits a single character to the output segment. It requires two parameters, the segment index, and the character to be transmitted. Thus, for example:

```
OPCH (index2, #x)
```

has the effect of transmitting the single character "x".

If the character C/R is transmitted the current image is considered to be terminated. If the implementation is line image based, the text message may not be transmitted until the C/R character is received. If it is desired to embed a carriage-return within an image, octal 0215 should be used as the value of the character. In this case the character will not be recognized as the image terminating character. Instead it will be treated as a normal character. Since bit 8 of the character is the parity bit, it will be reset by the output primitive.

10.6.2 The PUTCH Primitive

The PUTCH primitive stores a single character into a string. It requires two parameters:

1. The address of a CAP
2. The character to be stored.

The form of the PUTCH primitive is:

```
PUTCH (@CAP,#$) .
```

If the CAP is as shown in Figure 10-1 then the character "$" would replace the character "i". The PUTCH primitive does not increment the character index. The character count in the first word of the string is unaffected.

10.7 Compiler Output Facilities

The procedures described below provide the facilities used by the compiler, and available for users, for output to external segments.

10.7.1 The SO Directive

All compiler output functions operate by use of the segment index contained in the variable SOUNIT. The SO directive may be used to direct all compiler output to a specified segment. The form of the SO directive is:

```
SO character-string .
```

The SO directive uses the function FNAME to create the segment-name string and passes this string to the OPENF primitive. The returned index is stored in SOUNIT.

10.7.2 The OBREAK Directive

The OBREAK directive closes the output segment whose index is given by SOUNIT, and redirects the compiler output to segment index zero. OBREAK requires no parameters.

10.7.3 The OUTST Function

The OUTST function transmits a MINT formatted string to the current output segment as designated by SOUNIT. It requires as a parameter the address of the string to be output. The form of the OUTST function is:

```
OUTST('This is a string') .
```

10.7.4 The OPINT and OPINTD Functions

The OPINT function converts an integer to character form and sends the characters to the output segment designated by SOUNIT. The OPCH primitive is used to transmit the characters. If the conversion base is decimal (See Section 10.7.6) the number is treated as a signed integer and, if the integer is negative, the digits are preceded by a minus sign. In all other cases the digits are preceded by a blank character. The OPINT function requires one parameter which is the integer to be converted. Thus, its form is:

```
OPINT(arithmetic-expression).
```

The OPINTD function converts its argument as an unsigned integer. No space or minus sign is output preceding the digits. Use of OPINTD is appropriate if, for example, address values are to be output. Its form is:

OPINTD(arithmetic-expression) .

The OPINT and OPINTD functions convert the supplied number according to the information provided by the SETOPP and SETBSE functions, which are described in the two following Sections. If the number of digits required for the number exceeds the available field width the first printed digit will be a question mark. The remaining field is filled by the least significant digits.

10.7.5 The SETOPP Function

The SETOPP function establishes the format of numbers output by the OPINT and OPINTD functions. It requires two parameters:

1. Pad character (PAD)
2. Field width (WIDTH).

The pad character is used to left fill the printed number. The field width is the total number of digits and pad characters to be output. A width of zero provides variable width output. In addition, the variable CWIDTH is set to the width actually used on each OPINT or OPINTD reference. The width used when WIDTH has been set to zero is the width required to display the digits. Thus,

SETBSE(#0, 0, 10), OPINTD(123)

will display 123 without any leading or trailing blanks. After this operation the value of CWIDTH will be 3.

In addition to the number of characters specified, a blank or minus sign is output as the first character by OPINT. Thus,

SETOPP(#0,5), OPINT(123)

yields 00123

and

SETOPP(# ,7), OPINT(MINUS 4567)

yields - 4567 . The OPINT and OPINTD functions always reset the PAD character to zero (#0) after printing.

10.7.6 The SETBSE Function

The SETBSE function resets the conversion base used by OPINT and

OPINTD. It also references SETOPP to set the pad and width values. It requires three parameters. The first two are passed to SETOPP. The last one is used to set the conversion base. The bases which are available are: 16, 10, 8, and 2. Thus, for example:

```
SETBSE(#0,4,16)
OPINT(255)
```

yields 00FF .

When first referenced, the compiler operates as though a

```
SETBSE(#0,5,10)
```

had been performed.

The identifier BASE contains the value of the current conversion base. It should only be modified through a reference to SETBSE.

10.7.7 The OPNL and OPFF Functions

The OPNL function causes transmission of a carriage-return (C/R) character to the current output segment as defined by SOUNIT. In Virtual Machine implementations which interface to facilities which require line images no data are actually transmitted to the segment until this character is received by the Virtual Machine. Thus, it is necessary to transmit C/R, usually by use of OPNL, after a series of one or more references to OPINT, OPINTD, or OUTST in order to cause a complete line of output. The OPFF function causes transmission of a form-feed (F/F) character, and thus is intended to cause a skip to a new page when printing is performed.

10.8 Closing of Segments

Both input and output segments are closed by the CLOSEF primitive which is described below. The primary I/O streams do not need to be closed.

10.8.1 The CLOSEF Primitive

External segments are closed by the CLOSEF primitive which requires one parameter. The parameter is the segment index of the segment to be closed. Thus, an example reference is:

```
CLOSEF(index1) .
```

An attempt to execute I/O functions using an index of a segment which is not currently open will result in a Virtual Machine error. However, it is not a requirement that each index returned by OPENF be unique. Each new index must differ from any of the currently open indices. Thus, indices may be reused. For this reason it is good practice to reset to zero any variables in which an index was stored when a CLOSEF is performed on that index.

10.9 The String Matching Primitives

There are two primitives which carry out string matching operations. The MATCH primitive compares a *key* string against a string addressed by a CAP. The DICMATCH primitive compares a series of linked key strings against a string addressed by a CAP. The DICMATCH operation is carried out by repeated application of MATCH.

10.9.1 The MATCH Primitive

The MATCH primitive provides the means of determining whether or not a given string contains a specified sub-string. MATCH expects two input parameters: the address of a key string and the address of a CAP. A character by character compare is carried out starting with the first character of the key string and the character pointed to by the CAP. Successive characters in each string are compared until a mismatch is found or until an end-of-string condition occurs. If the end of the key string is reached the string match condition is indicated by obtaining a Boolean true on the stack. The CAP character pointer is reset to point to the character after the last matched character. Otherwise, a Boolean false is obtained and the CAP is unchanged. If the key string is longer than the sub-string to which the CAP points no match is possible and a false is always returned. For example:

```
VAR STR1: 'This string is long enough'
VAR CAP1: @STR1
VAR CHRNO: 5
VAR KEYSTR: 'string'
MATCH(@KEYSTR, @CAP1)
```

will return a true result, and CHRNO will be set to 11.

10.9.2 The DICMATCH Primitive

The DICMATCH primitive allows matching a list of key strings against a CAP-addressed sub-string. DICMATCH applies the MATCH operation using each of the key strings in the list until a match is found or until the end of the key string list is reached. Three parameters are required for DICMATCH. These are: the address of the list start pointer, the address of the list end pointer, and the CAP address. If a match occurs, DICMATCH returns the address of the record prior to the matched record and the CAP is updated to point to the character after the last matched character. If no match occurs the address of the list end pointer is returned and the CAP is unaffected.

10.10 The COMPILE Function

Source text is compiled into object text by means of a reference to the COMPILE function. The COMPILE function expects as its single argument the address of a string, which is the source text. If COMPILE is referenced in normal input, the compiler will compile the addressed string and then return to the point of reference just as is done with any other function. Therefore, text may be constructed as a string and then compiled whenever this operation is required. For example, a string could be obtained from an external segment and compiled by means of:

```
VAR INBUF: RESERVE 42
VAR MAXL:80
VAR INUNIT:0,
        OPENF(1,'sourcetext') -> @INUNIT,
        GETSTR(INUNIT,MAXL,@INBUF),
        COMPILE(@INBUF) .
```

The compiler's main processing loop consists of repeated application of COMPILE to the source input strings.

10.11 Problems

10.11.1 Problem 1

Define a "dictionary" list which contains records made up of a field for the

dictionary word and a field for the definition of the word. Write a function which searches the dictionary for a given word and prints the definition if a match occurs.

10.11.2 Problem 2

For the list used in Problem 1 write a function which accepts a list of keywords and prints each dictionary entry whose definition field contains any one of the keywords.

10.11.3 Problem 3

Using the MATCH primitive, write a function which performs according to the specification of the DICMATCH primitive.

10.11.4 Problem 4

Write a set of functions to copy an external segment and optionally display it on the primary (segment zero) I/O device.

10.11.5 Problem 5

Write a function which compiles the source text contained in an external segment. Consider at least two alternative methods of implementing this function.

11. The Syntax Analysis System

11.1 Introduction

The syntax analysis system (M-TRAN) is a product for interpreter or compiler writers (or indeed the implementors of any language or dialog system) permitting the definition of the syntactic structure (or grammar) of a language. This obviates the need to write a program to analyze the language text. Thus, implementations may be more compact, more uniform, and more easily managed.

The system permits generation of VM(M) Virtual Machine operations, or MINT text should the user wish this. This gives an implementation the same portability as MINT itself. The user is entirely free to bypass this of course, and generate any form of text he wishes.

The system is a top-down, fast-back analyzer which uses a generalization of Backus Naur Form (BNF) for syntax definition. The main generalization is the fact that normal MINT functions may appear in the phrase definitions. Since M-TRAN is written in MINT it is an extension of the basic system. M-TRAN makes use of the CLASS directive to create the class PHRASE, which is the basic component of the system.

11.2 Phrase Structure Analysis

The syntax analyzer operates on BNF definitions which are implemented in the form of MINT functions. A BNF definition must first be introduced in class PHRASE, and then defined as described below.

11.2.1 Phrase Introduction

Identifiers in class phrase are introduced in exactly the same manner as identifiers in other classes. Thus,

```
PHRASE const
```

introduces the identifier const in class phrase.

11.2.2 Phrase Definition

A phrase definition consists of the phrase identifier followed by ::=, and then the phrase elements which direct the parsing. The phrase entry must always begin with the ::=. This is followed by one or more alternatives separated from each other by //. Thus, phrase definition takes the form:

```
name ::= ...../...../.....
```

The alternatives themselves consist of one or more functions, which are executed in sequence. These functions may be any function, but will generally be either other phrases, or parsing functions. Thus,

```
item ::= ident // number .
```

sets, and defines the characteristics of, the phrase identifier item. After completion of a set of phrase definitions, the directive PEND must be referenced to return the compiler to normal text processing mode.

M-TRAN provides much more flexibility than formal BNF because of the ability to execute any function and even pass parameters using all of the usual MINT tools during the parsing itself. For example, consider the following:

```
item ::= ident idgen // number constgen //

OUTST('syntax error') .
```

The phrase definition for item is established as follows: ::= sets the phrase entry. An attempt is made to match the input string against the ident definition. If the parse is successful the next function in the current alternative is executed, i.e. idgen which could generate object text appropriate for an identifier. If on the other hand the parse is unsuccessful the next alternative is tried beginning at the //. In this case an attempt is made to match the input string against the number definition. If successful, constgen is referenced. Constgen may generate object text for a constant. Parsing would then terminate in this alternative. If the second alternative also fails, the last alternative is tried. This is a function reference to output an error message.

As another example, consider the text required for analysis and evaluation of a simple arithmetic expression. The syntax definition for the expression is:

ADD= (-)<number> [(+<number>|−<number>)]

where ( ) encloses optional symbols, | delimits alternatives, < > encloses syntactically defined objects, and [ ] indicates that repetition is allowed. Thus, a valid use of ADD is:

ADD=10+20+4 .

The text which implements the ADD mechanism is given below:

```
.
VAR NUM:0
FN  clr:ENTRY, WHILE NE 0 START REPEAT, EXIT
.
PHRASE const
PHRASE moredgts::=const//NUM,0 ->@NUM
       const   ::=BLANKS,DIGIT NO FAIL, NEXTCH-#0+NUM*10
                  ->@NUM, moredgts
PHRASE expr
PHRASE subexpr ::='+'$$,+expr//'-'$$,-const,subexpr//';CR'$$
       expr    ::=const,subexpr
PHRASE frstexpr::='-'$$,-const,subexpr//const,subexpr
PHRASE statm   ::=0,BLANKS,frstexpr//OUTST('Invalid
                  . syntax.'), OUTST(' Last valid number: '),
                  OPINT, clr, GO FAIL
PHRASE analyze ::='='$$,statm,OUTST('Result: '),OPINT, LOSE,
                  //OUTST(' Try again.')
PEND
.
DIR ADD:ENTRY,BLANKS,analyze,OPNL,REF.,EXIT
```

In this example, analysis and evaluation are initiated by a reference to ADD. The directive ADD references the phrase analyze which first tests for the presence of "=". If this symbol is present in the input stream the phrase statm is referenced. If "=" is not present this alternative in analyze fails and the second alternative is tried. This alternative simply generates the output text "Try again." and control returns to the ADD directive. The phrase statm obtains a zero on the stack as an indicator and then tests for a valid first expression following the "=". The phrase frstexpr allows either -number or number followed by a subexpression. The phrase const both tests for a number and accumulates the value of the number on the stack. The phrase subexpr tests for the form plus or minus followed by an expression, or a C/R character. The expression form is tested in expr

which looks for a number followed by a subexpression. If the phrase subexpr successfully matches the C/R character then control returns to frstexpr with the accumulated arithmetic result on the stack. If expr, subexpr, or const fail control returns through frstexpr to statm which tries the next alternative. This results in the message "Invalid syntax...". Since the failure return will have caused subexpr not to have accumulated any results, the item on the stack will be the last item successfully evaluated by const. This number is written by statm to provide an indication of the point of failure. The function clr resets the stack to before the zero item which was obtained by statm. The failure exit is then taken from statm which causes analyze to try the next alternative. This just writes the "Try again." message and exits to ADD.

The close correspondence between phrases and BNF definitions should be evident. The directive // corresponds to the vertical bar symbol in BNF. Phrase names are not enclosed in < and > and basic symbol strings must be enclosed in quotes and operated upon by the function $$.

11.3 Parsing Functions

The general use of parsing functions is the basis of the flexibility of the phrase structure analyzer. For example, a string matching function ($$) is provided and is used as follows:

```
pqr ::= $$('abc') // $$('def') .
```

In this example an attempt is made to match the string abc. If that fails an attempt is made to match def. If that fails the entire phrase fails. More generally, $$ is allowed to operate on any string obtaining expression. Thus in

```
$$(x EQ 1 CHOOSE ('pqr',string1))
```

an attempt is made to match the string returned by the expression in parentheses, i.e. pqr or string1 depending on the value of x. In this way (among others) the parsing can be made context sensitive.

All parsing functions are perfectly standard functions with the exception that they must drive the so-called failure mechanism on a non-match. This may be illustrated by defining the phrase for digit. It would be inefficient to write:

```
digit ::= '0'$$ // '1'$$ // '2'$$ ...
```

The phrase is better implemented as a function:

```
FN digit:ENTRY
        CHAR GE #0 AND CHAR LE #9
        NO FAIL,
        EXIT .
```

FAIL is the label of the M-TRAN failure mechanism; phrases implemented as functions must transfer control to FAIL on failing to find a match condition. The failure mechanism automatically causes parsing control to return from a rejected trial, (i.e. a function which jumped to FAIL; or a phrase all of whose alternatives have failed) in order to pass control to the next alternative of the referencing phrase.

11.4 Optional Elements

The directive OPTION may be placed at the end of a phrase definition. Its meaning is that the alternative is optional, but does not cause matching on its own. Thus, the definitions

```
p1 ::= 'abc'$$ p2
p2 ::= 'def'$$ OPTION
```

allow the strings abc and abcdef but would disallow the string def on its own.

11.5 Phrase Function Usage

The following Sections indicate a few of the ways in which phrase functions may be used.

11.5.1 Phrase Functions

A phrase may return a value. For example one might design a phrase called integer which would parse source text for the correct syntactic form of an integer. But, in addition, this phrase could compute the value of the integer while each digit was being parsed. The final computed value could be left

on the stack, to be used at will by the context that used the integer. A more complex application might be for constant subscript evaluation in a Fortran equivalence or data statement.

11.5.2 Phrase Parameters

Phrases are permitted to have parameters. For example, Fortran contains in several contexts lists of syntactic entities separated by commas and enclosed in parentheses. We could therefore define a phrase to express this, called braklist, with one parameter (the syntactic entity), and use this phrase in each context. Thus, references would be of the form:

```
braklist (@statnum)
```

for a list of statement numbers in a computed go to, and

```
braklist (@param)
```

for a parameter list context. The use of @ is necessary to pass the address of the phrase parameter. Were it not used, these phrases would be activated at the point of reference with unintended results.

11.6 Listing of M-TRAN

Below is the listing of the source text which implements M-TRAN.

```
.
.   BNF TOP-DOWN FAST-BACK ANALYSER
.   ------------------------------
.
.  @Copyright    D. F. Hendry.
.
ICL$                                . introduce internal identifiers
VAR PHREC:                          . current phrase record
VAR FAILX:0                         . current failure jump
VAR LINKP:0                         . saved link pointer
VAR COL:0                           . saved CAP index
VAR PHLIST:0,&(@PHLIST ADIFF @PHREC).  phrase record push-down list
.
VAR FEXITL:0                        . compile time failure jump list
VAR PHLAG:0
.
FN  SETPHR                          . phrase setting action
CLASS PHRASE:SHUNT,FNGEN,SETPHR     . introduce phrase class
.
```

```
.   RUN-TIME PARSING ALGORITHM
.
FN  BOB:ENTRY                          . beginning of branch
    PUSHDREC(@PHREC,@PHLIST)           . push down current phrase record
    ->@FAILX, ->@LINKP,                . fail exit list pointer and link
    CURCOL ->@COL,                     . current CAP index
    EXIT
.
LAB EOA:                               . end of an alternate
    SLKP(LINKP),                       . set link pointer
    POPUPREC(@PHREC,@PHLIST)           . pop-up higher level phrase record
    EXIT
.
LAB FAILUP:                            . failed all alternates
    SLKP(LINKP),                       . set link pointer
    POPUPREC(@PHREC,@PHLIST),          . pop-up higher level phrase record
    LINKP EQ 0 THEN <EXIT>             .
LAB FAIL:                              . failed one alternate
    COL ->@CURCOL,                     . and input string status
    GO VAL(DUP(FAILX-1) ->@FAILX),     . adjust fail pointer, try next alt
.
FN  $$:ENTRY                           . string matching function
    MATCH(,@CURCHS) THEN<EXIT>         . successful match
    GO FAIL,                           . drive failure mechanism
PAGE
.   COMPILE TIME FUNCTIONS
.
FN  FAILMECH:ENTRY                     . construct fail list + phrase end
    PUSHD(,@FEXITL)                    . last jump addr to list
    COMPILE(',GOEOA,')                 . end of alternate jump
    WHILE FEXITL NE 0 START            . empty list at bottom of phrase
        STORE1(POPPEDUP(@FEXITL))
    REPEAT
    COMPILE('>')                       . close forward ref (list addr)
    0 ->@PHLAG
    EXIT
.
DIR PEND:ENTRY                         . terminate phrase
    PHLAG EQ 1 THEN<
    FAILMECH(@FAILUP)>                 . last failure jump
    EXIT
.
SETPHR:ENTRY                           . phrase setting action
    SETPROG                            . set PROG state
    CUR REF PEND ->@CUR,               .
    1 ->@PHLAG
    EXIT
.
MACRO ;:=:'ENTRYGLKPBOB(<)'            . phrase entry code
.
DIR //:ENTRY                           . next alternate
    COMPILE(',GOEOA,')                 . successful jump
    PUSHD(PLOC,@FEXITL)                . location of new alternate
    EXIT
.
DIR OPTION:ENTRY                       . creates phrase as optional
    PHLAG EQ 1 THEN<
```

```
    FAILMECH(@EOA)>                 . last failure jump successful
    EXIT
RCL$                                . disallow internal identifiers}
```

12. MINT Techniques and Examples

12.1 Introduction

This Chapter contains an introduction to standard techniques for writing and analyzing MINT text, and some examples of MINT text. The examples are arranged in order of increasing complexity.

12.2 Entering Text

There are several ways in which text may be entered into a MINT system. Generally, if the text is to be retained it should be placed in a file and then referenced using the SI directive (Section 10.5.1). The means of entering the text into a file, and of editing files, will depend on the facilities of the system in which MINT is implemented. If the system does not have its own text editing system, the editor given later in this Chapter (Section 12.6) may be used. The advantage of entering the text into a file is that, in most implementations, the current state of the MINT system is lost if the VM must be restarted. In addition, when new text is being written it will often be the case that the text will be revised several times before the intended results are obtained.

12.3 Translation and Manipulation of Text

Once text has been entered into a file, it may be provided as input to the system by means of the SI directive. Any text at all may be included in a file which is referenced by SI. Thus, the text may be composed of such things as variable declarations, function definitions, macros, directive references, or NOW . . . ! sequences. Of course, any such sequences may be typed directly into the system with exactly the same effects. A standard way of writing text to accomplish some new result is to enter the text in a file, call the VM and load the compiler, reference the file by means of SI, establish any initial conditions or initialize diagnostic aids such as tracing or trapping, and then

reference the appropriate directives or functions in the text. If the intended results are not obtained, individual procedures may be replaced or variables changed. Such changes should be done with care so that any changes which are intended to be permanent can be remembered and applied to the copy of the text which is retained in a file. When a significant amount of text is working as intended, the copy of VSTORE with this text included may be written out to a file using the CDUMP (Section 6.6.5) primitive. This VSTORE copy may then be reloaded subsequently so that facilities may be developed incrementally. The CDUMP operator may also be used to preserve the current state of VSTORE so that work may be resumed at that point at a later time.

12.4 Analysis and Diagnostic Techniques

The purpose of this Section is to provide general guidance about the use of MINT analysis and diagnostic mechanisms. The general intention of these mechanisms is to make it possible to cause the state and operation of the MINT system to be entirely visible. By this we mean that it is possible to record or display any information in VSTORE or any operation of the Virtual Machine. Thus, if any state or behavior is not understood it can be displayed for detailed analysis. This potential visibility is essential for predictable and efficient problem analysis.

Generally, we emphasize analysis rather than diagnosis as a working method. If MINT text is produced by careful writing and analysis of each elementary component then correct operation is to be expected and there will only be exceptional need for subsequent diagnosis.

We tend to be opposed to "iterative refinement" as a means of writing well-constructed text. If the current attempt does not perform correctly it is generally advisable to rethink the entire task and make a new start. The compactness of MINT text tends to enhance the efficiency of this approach.

12.4.1 Analysis of Data Structures

Frequently the most important aspect of analysis is the understanding of the transformation of the data structures. The most direct means of understanding these transformations is to write procedures which display each structure, or relevant portions of the structure. These procedures will naturally be written as the data structures are being defined. If lists of items or records are used, procedures which selectively display the list items will likely be needed. If the task that is being written uses data that are being

received from a user or directly from a communications line, it is appropriate to write procedures which will simulate the data input environment. This simulated environment will permit controlled experiments as the text is written. This is particularly important for text which is to be used for "realtime" control of equipment or processes.

If functions are used to access the items in a record (as discussed in Section 7.2.5) then it is a simple matter to place "intercept" references in the accessing functions so that the sequence of operations on any item in a record is easily followed. This intercept notion can easily be generalized to operate on any reference since a variable name may be changed to a function which obtains or stores the value, but which also provides display information. For example, the variable var1 could be replaced by:

```
VAR xvar1:0
VAR pvar1:@xvar1
FN @var1: ENTRY, OPINT(VAL(DUP(pvar1))),
                 OPNL, EXIT
FN var1 : ENTRY, VAL(@var1), EXIT
```

Then, a reference to var1 would obtain the value of xvar1 after displaying its value, and a reference to @var1 would obtain the address of xvar1. Again, the value of xvar1 would first be displayed. After these changes any text which referenced var1 as a variable would yield the expected results but in addition would produce any additional actions which were written into the functions.

In simple cases where it is only necessary to inspect sequential locations in data-space the following procedure could be used:

```
FN ovec:ENTRY,0 ->@TEMP, WHILE DUP(-1) GE 0
        START
        <=>, DUP, OPINT(VAL(FROM TEMP))
        ADV(@TEMP), LOSE(TEMP/10), DREM EQ 0
           THEN<OPNL>, <=>,
        REPEAT, LOSE(TEMP/10), DREM NE 0
           THEN<OPNL>,
        LOSE, LOSE, EXIT
```

The function ovec expects the address of the vector and its length as arguments. Thus, execution of

```
ovec(@v1,20)
```

would display the 20 words starting at @v1.

12.4.2 Analysis of Generated Object Code

It is frequently of interest to see what actual Virtual Machine instructions have been generated or are executed in a specific context. The simplest means of displaying generated instruction sequences is the D$ directive as described in Section 8.6.1. It may happen that the compiler has generated unexpected sequences due to shunt or precedence effects. D$ will reveal such effects. If it is the execution behavior of a sequence that is of interest, T$ may be used to cause display of each Virtual Machine instruction and current operand stack values as the instructions are executed. The T$ directive is described in Section 8.6.2. Since individual operation codes may be traced, compact but informative tracing is possible. The TRAP operator may also be used for other analysis purposes as shown in the example in Section 12.7.

In case it is necessary to determine the names, and addresses where set, of some identifiers, the LV$ directive may be used. This directive, as explained in Section 8.6.3, provides information about the current dictionary entries.

12.4.3 Instruction Emulation

If it is necessary to perform analysis based on single instructions the most selective and efficient technique is to use the EMULATE operator. The EMULATE operator allows the replacement of a single Virtual Machine instruction by a normal procedure written in MINT. Previously undefined operation codes may also be defined by means of EMULATE. There are two obvious uses of EMULATE. The first is to investigate behavior of executions as a function of individual instructions. For example, a store into a particular VSTORE location could be occurring unexpectedly. Replacement of the store operator with emulation text which checks for the specific address will detect the source of the event immediately. Second, the use of possible new primitives may be investigated more easily by first writing them as MINT procedures prior to incorporating them into the Virtual Machine. The use of EMULATE is fully described in Section 6.6.2.

12.5 A Simple Calculator

This example shows how a basic desk calculator may be written. The procedures which are used are written as directives so that they will execute immediately when referenced.

12.5.1 The EVAL Directive

This directive will evaluate any valid MINT expression and display the resulting value. The directive is:

```
DIR EVAL: ENTRY,
        IPAR,
        OPINT( ), OPNL, EXIT .
```

The directive operates as follows: IPAR (See Section 8.2) reads the next expression from the current input source, evaluates it and leaves the result on the operand stack. OPINT converts and outputs this result. OPNL outputs a carriage-return, thus completing the current output image. An example usage is:

```
EVAL (2+2)
```

with the resulting display:

```
00004 .
```

Another usage is:

```
VAR X1:25
VAR X2:30
VAR X3: MINUS 12
EVAL ((X1+X2+X3)/6+15)
```

which will yield the display:

```
00022 .
```

12.5.2 An Addition Directive

This directive provides a simple routine which will sum a column of numbers. Whenever a non-numeric character is input the routine will print the current total and terminate. The text for the directive is:

```
DIR ADD: ENTRY,
        0, WHILE READ, BLANKS, DIGIT START
                +ININT,
        REPEAT
        OUTST(' Total: ') OPINT()
        OPNL, REF . , EXIT .
```

The directive will carry out the following operations: First it initializes the

accumulation by obtaining a zero on the stack. Then it starts a loop which reads the next image (READ), removes any leading blanks (BLANKS), and tests for the presence of a digit (DIGIT). If a digit is present it converts the input to an integer (ININT) and adds it to the current top-of-stack (+). If a digit was not found the REPEAT loop terminates. At this point the message which gives the current total is displayed. Then a reference to the period directive is made so that any remaining text in the current input line is discarded, and the procedure terminates. Thus,

```
ADD
        10
        20
        45
        9
        T
```

will result in the display:

```
Total: 00084 .
```

A slightly more elaborate version of this directive is shown below. This version carries out the accumulation as before, but will also display sub-totals as the main total is being accumulated. A sub-total is requested by typing S. The final total is requested by typing T. Any other non-numeric input is ignored. The procedure is as follows:

```
DIR ADD: ENTRY,
0, WHILE READ, BLANKS, CHAR NE #T START
        DIGIT THEN <+ININT ELSE
        CHAR EQ #S THEN <OUTST(' Sub-total: ')
        OPINT(DUP),OPNL> >
        REPEAT,
        OUTST(' Total: ') OPINT()
        OPNL, REF . , EXIT
```

Most of this directive should be clear from inspection. However, notice that the value of the sub-total is displayed by means of the expression OPINT(DUP). The reference to DUP obtains an extra copy of the current top of stack so that the current total is still on the top of stack after the OPINT has removed the top object and displayed it as an integer.

12.6 Text Editing Directives

This example shows a fairly simple use of MINT for text manipulation,

and provides a useful facility if MINT is used in a system which does not itself provide an effective text editing mechanism. It will be seen in this example that the use of the IPAR mechanism allows the recognition and evaluation of expressions without special effort. In addition, the use of the string matching primitives makes the editing directives which require matching of strings or substrings particularly simple to write.

Since these editing directives form an extension of the MINT system, all general MINT facilities are also available during editing. For instance, directives, functions, or macros may be written at any time to simplify a specific editing task. The basic procedures given below provide building blocks for creation of a wide range of editing structures. The object text uses 1197 words of VSTORE for instructions and 262 words for data.

The following is the complete file of the editing system in the exact form which is used as input to the compiler. The detailed documentation is kept in the file so that the editing system is entirely self-contained.

12.6.1 Listing of MINT Editor Source Text

```
. MINT  TEXT  EDITING  FACILITIES
.
. @Copyright   1980 M.D. Godfrey, H. Hermans
.
. The directives below provide a basic text editing
. mechanism within MINT. In order to initiate editing, a
. reference to the directive EDIT is required. EDIT expects
. a string parameter which is the name of the input file.
. Editing is terminated by a reference to the END directive.
. END also expects a string parameter which is the name of
. the file into which the edited output will be written.
.
. Thus, editing is initiated by:   EDIT infile
.
. and terminated by:               END outfile
.
. If END is used without a parameter, editing is
. terminated without writing any output.
.
. The editing process operates on sequential line-structured
. files. Lines may be inserted or replaced, or substrings
. within a line may be replaced by a new substring. The
. editor may be requested to go to a specified line in the
. file, or may be requested to search each line until it
. reaches a line which contains a specified string.
.
. A pointer, the current line number, is maintained by the
. editor. When editing starts the pointer is at zero, which
. is before the first line. In this case, and in the case
. where the line at the current position has been deleted,
```

```
. there is no current line image. In this case, the editor
. may be considered to be between lines.
.
. The editor treats files as forward sequential. It reads
. the input file in a forward direction, copying the lines
. and any changes into a scratch file. When the end-of-file
. is reached on input, or when a request is made which
. causes positioning to a line above the current position,
. the current forward copy is completed, the files are
. closed and the scratch file is used as input. At this
. point a second scratch file is used for output. At the
. next end-of-file condition, the files are closed and the
. two scratch files are exchanged so that the previous
. output becomes the input. Finally, the END statement
. causes current copying to be completed and the output
. scratch file to be copied to the specified user file.
.
. The editor has two "modes".  In "input" mode, which is
. initiated by the / directive and terminated by a blank line,
. the editor simply reads the text typed and stores it in the
. file at the current file position.  In "edit" mode the
. directives described below are available:
.
. 1. Positioning Directives.
.
.      G expr
.      G +expr
.      G -expr
.
. This directive causes positioning to a line number in the
. file. If the first form is used (G expr) the line number
. is the result of the evaluation of 'expr'. If the second
. form is used the line number is the current line number
. added to the result of the evaluation of expr. If the
. third form is used the line number is the current line
. number minus the value of expr.
.
. Here, and in all other directives, 'expr' is any
. expression which may be evaluated by means of the IPAR
. mechanism.
.
.      N expr
.
. This is equivalent to  G +expr. N with no parameter is
. treated as N 1.
.
. 2. String Search Directives.
.
.      F string
.      F,expr string
.
. This directive searches through the lines in the file
. until it finds a line which contains the text which was
. supplied as the string parameter. The matching of the
. supplied string and each line is done by treating the line
. as one string and the parameter, starting after the blank
. following the directive, as the other string. Thus,
```

```
.
.      F     X
.
. would match a line which contains 4 blanks followed by an
. X in column 5. If it is desired that the match start in
. some column other than column 1 of each line image, the
. form F,expr should be used. Expr is an expression which
. yields a value which is taken as the starting column for
. the match. Thus,
.
.      F,10 xx
.
. would match on the first image which contained xx starting
. in column 10.
.
.      L string
.      L,expr string
.
. This directive searches through the lines in the file
. until it finds a line which contains a match on the
. supplied string anywhere within the line. Thus, unlike the
. F directive, the match attempt is applied using the entire
. line, then the substring starting in column 2, then column
. 3, etc. Thus,
.
.      L cat
.
. would locate the next line which contains the substring
. cat anywhere in the line. If it is desired that the match
. should start in a column other than column 1, the form
. L,expr should be used. In this case the matching in each
. line will start at column expr and continue to the end of
. the image.
.
. 3. Line Insert Directives.
.
.      I string
.      I,expr string
.
. This directive inserts the text string following the
. current line. If the first form is used the text is copied
. from the position following the blank after I to start in
. column one of the created line image. Thus, above, string
. would start in column one of the new image. If the second
. form is used, the copy is to the column given by expr.
. Thus,
.
.      I,14 text input
.
. would create a new image composed of the text, text input
. which would start in column 14.
.
.      IB string
.      IB,expr string
.
. This directive inserts the text string before the current
. line.
```

```
.
.      R string
.      R,expr string
.
. This directive inserts the text string in place of the
. current line.
.
. 4. Line Delete Directive.
.
.      D expr
.      D expr1 expr2
.
. This directive deletes lines. If no parameters are used,
. it deletes the current line. If one parameter is used it
. deletes expr lines starting with, and including, the
. current line. If two parameters are given, it skips expr1
. lines and then deletes expr2 lines.
.
. 5. Text Change Directive.
.
.      C /string1/string2/
.      C,expr /string1/string2/
.
. This directive acts on the current line. It searches for
. the string string1. If this string is found it replaces
. the found string by string2. The preceding and following
. parts of the line are left unchanged. If C,expr is used
. the search for string1 starts in column expr.
.
. 6. Line Display Directive.
.
.      P expr
.      P expr1 expr2
.
. This directive displays the lines given by the parameters.
. The parameters have the same meaning as with the D
. directive.
.
. 7. Miscellaneous.
.
.      LN
.
. This directive displays the line number of the current
. line.
.
.      RV
.
. This directive causes the current input and output files
. to be closed, and the input file to be reopened for input.
. This has the effect of discarding any changes made during
. the current pass through the file.
.
.      V ON/OFF
.
. This directive controls 'verify' mode. If verify mode is
. on the directives L, F, C, and N cause the current line to
. be displayed before they exit. If verify mode is off this
```

```
. display is not done.
.
.      LNUM ON/OFF
.
. This directive controls the printing of the current line
. number preceding the printing of each line image. If LNUM
. is on, the line number is displayed in columns 1 through
. 6. Column 7 will contain :. If LNUM is off the image is
. displayed without any line number.
.
. Introduction of all editing directives.
.
DIR EDIT       . entry into editing mode.
VAR EDDIR:0    . list of EDIT identifiers.
        BLOCK
DIR END        . terminate editing.
DIR C          . change.
DIR D          . delete lines.
DIR F          . find.
DIR G          . go to line.
DIR I          . insert.
DIR IB         . insert before.
DIR L          . locate text.
DIR LN         . print current position.
DIR LNUM       . set or reset line numbering mode.
DIR N          . move to next line.
DIR P          . print text.
DIR R          . replace a line.
DIR RV         . revert current changes.
DIR V          . set or reset verify mode.
DIR /          . for input mode.
.
. Variables for use with V and LNUM.
.
VAR ON:1       .
VAR OFF:0      .
VAR ALL:9999,
ICON EOFF 256,
ICON MSI$ 1, ICON MSO$ 2,
VAR CIFT:MSI$, VAR COFT:MSO$,
.
       BLOCK   . Restrict further introductions to within
               . editing.
.
VAR DATE EQV 12 . define location of current date.
ICON CRC 13     . define C/R character value.
VAR LNP:0      . 1-> print line numbers.
VAR VERI:1     . verify mode flag.
VAR COF:0      . output file index.
VAR CIF:0      . input file index.
VAR LNO:0      . current line number.
VAR CHGS:0     . 0-> no changes this pass.
VAR STCOL:1    . start column for I, IB, F, L, C.
VAR SKNT:0,    . lines to skip.
.
VAR NAM1:'EDIT_A.TMP', VAR NAM2:'EDIT_B.TMP', .
VAR OUTF:@NAM1, VAR INF:@NAM2,     . set file pointers.
```

```
VAR OUTFN:0    . pointer to output file.
.
VAR BCAP:<
VAR BCNO: 0, >
VAR BFR: RESERVE 68,
VAR LSTR:<>, VAR LBFR:0, RESERVE 39,
.
.      common subroutines.
.
. function to set changed indicator.
FN CHG:ENTRY 1->@CHGS, EXIT
.
. check for non-terminal char.
FN EOSTR: ENTRY CHAR NE 13 EXIT
.
. process any modifier field.
FN MODFS: ENTRY, CHAR, ADVCH, EQ #, THEN <IPAR, ADVCH ELSE 1>
       ->@STCOL, EXIT
.
FN CKIOPN: ENTRY,
       CIF EQ 0 THEN <CLOSEF(COF),OUTF,INF,->@OUTF,->@INF,
           OPENF(CIFT,INF)->@CIF,OPENF(COFT,OUTF)->@COF,
           DUP(0)->@LNO,->@CHGS,>
           EXIT
.
. output current image to file.
FN OUTCIM: ENTRY, CKIOPN,
       BFR NE 0 THEN<SOUNIT, COF->@SOUNIT,
      OUTST(BCAP),->@SOUNIT,>
       EXIT
.
. move next input image to current image.
FN MOVENI: ENTRY
       0->@BCNO,
       STCOL, WHILE DUP GT 1 START
           PUTCH(@BCAP,# ),ADV(@BCNO),-1,
       REPEAT, LOSE,
       CURCOL, WHILE DUP NE VAL(CURCHS) START
           DUP ->@CURCOL, PUTCH(@BCAP,GETCH(@CURCHS)),
           ADV(@BCNO),
           ADVCH,+1, REPEAT, LOSE,
       BCNO EQ 0 THEN<PUTCH(@BCAP,CRC),ADV(@BCNO)>
       BCNO->BCAP, ADV(@LNO),
       EXIT
.
. store next input image in file.
FN STORNI: ENTRY, CKIOPN,
       STCOL, WHILE DUP GT 1 START
           OPCH(COF,# ),-1,
       REPEAT, LOSE
       GETCH(@CURCHS) EQ 0 THEN<OPCH(COF,CRC)>
       WHILE COF, DUP(GETCH(@CURCHS)) NE 0 START
           OPCH, ADVCH, REPEAT,
       LOSE, LOSE, ADV(@LNO),
       EXIT
.
. read next image from file.
```

```
FN RDNIM: ENTRY, CKIOPN,
       GETSTR(CIF,68,BCAP),
       BFR NE 0 THEN <0->@BCNO,ADV(@LNO),EXIT >
       OPNL, OUTST('End-of-file at line:'),OPINT(LNO),OPNL,
       CLOSEF(CIF),LOSE, DUP(0)->@CIF,DUP->@LNO,
       EXIT
.
. output current image and read next.
FN OUTRDN: ENTRY, OUTCIM, RDNIM(1), STCOL-1->@BCNO, EXIT
.
. print current image.
FN PRINT: ENTRY
       BFR NE 0 THEN<
           LNP NE 0 THEN<OPINT(LNO),OUTST(':')>
           OUTST(BCAP) >
       EXIT
.
. edit <input filename>.
REF EDIT:ENTRY
       OUTST('MINT Text Editor 1.5. Date: '),
       OUTST(@DATE), OPNL,
       OPENF(MSI$, FNAME)->@CIF, OPENF(COFT, OUTF)->@COF,
       DUP(0)->@LNO,->BCAP, CHG, SETBLOCK(@EDDIR),
       EXIT
.
. end <element name>
REF END: ENTRY,
       BLANKS, CHAR EQ 13 THEN <CLOSEF(COF), CLOSEF(CIF),
           OUTST('End of EDIT - no text filed.'), OPNL, EXIT>
       OUTCIM, SOUNIT, DUP(0)->@LNO, CIF,
       CLOSEF(COF),OPENF(CIFT,OUTF),
       FNAME,->@OUTFN,MSO$,DUP->@COFT,OPENF(,OUTFN)->@SOUNIT,
       WHILE DUP NE 0 START
          WHILE DUP(INCH(DUP)) NE EOFF START
           DUP EQ 13 THEN <ADV(@LNO)>
           OPCH(<=>SOUNIT),REPEAT,LOSE,
          CLOSEF(), REPEAT,
       CLOSEF(SOUNIT), LOSE, ->@SOUNIT,
       OUTST('End of EDIT - '),OPINT(LNO),
       OUTST(' lines filed in: '), OUTST(OUTFN),
       MSI$->@CIFT,MSO$->@COFT,
       OPNL, REF ENDBLOCK, EXIT
.
. / enter input mode.
REF /: ENTRY, MODFS,
       OUTST('Input mode at line '),OPINT(LNO),OPNL,
       WHILE EOSTR START OUTCIM, MOVENI, REPEAT
       OUTST('Edit mode at line '),OPINT(LNO),OPNL,
       CHG, EXIT
.
. LNUM directive.
REF LNUM: ENTRY, IPAR->@LNP, EXIT
.
. V directive.
REF V: ENTRY, IPAR->@VERI, EXIT
.
. LN directive.
```

```
REF LN: ENTRY, OPINT(LNO),OPNL,EXIT
.
. RV directive.
REF RV: ENTRY, 0->@CHGS, EXIT
.
. insert new image.
REF I: ENTRY
       MODFS,OUTCIM, MOVENI,CHG, EXIT
.
. insert new image before current image.
REF IB: ENTRY
       MODFS, STORNI, CHG, EXIT
.
. replace current image.
REF R: ENTRY
       MODFS,
       BFR NE 0 THEN <LNO-1->@LNO>
       MOVENI, CHG,
       EXIT
.
. common argument function for print, next, and delete.
FN DPOPT: ENTRY
       0->@SKNT,
       NOT EOSTR THEN <1, EXIT ELSE
           IPAR,
           EOSTR THEN <DUP->@SKNT,+IPAR>>
       EXIT
.
. print lines.
REF P: ENTRY
       DPOPT, WHILE DUP(-1) GT 0 START
           SKNT GT 0 THEN <SKNT-1->@SKNT, ELSE PRINT>
           OUTCIM, RDNIM, REPEAT,
       LOSE, PRINT, EXIT
.
. delete lines.
REF D: ENTRY
       DPOPT, WHILE DUP(-1) GT 0 START
           SKNT GT 0 THEN <SKNT-1->@SKNT,OUTCIM,
               ELSE LNO-1->@LNO, >
           RDNIM, REPEAT, LOSE,
       0->BCAP, CHG, EXIT
.
. go to next line.
REF N: ENTRY
       DPOPT, WHILE DUP(-1) GE 0 START
           OUTCIM, RDNIM(1) EQ 0 THEN<LOSE, EXIT>
           REPEAT,
       LOSE,
       VERI NE 0 THEN <PRINT>
       EXIT
.
. go to line #.
REF G: ENTRY
       BLANKS, CHAR EQ 13 THEN <0, ELSE CHAR EQ #+ THEN <
           NEXTCH, LNO+IPAR ELSE CHAR EQ #- THEN <
             NEXTCH,LNO-IPAR, ELSE IPAR >>>
```

```
       DUP LT LNO THEN <
           CHGS EQ 0 THEN <CLOSEF(COF), CLOSEF(CIF),
             OPENF(COFT,OUTF)->@COF,
             OPENF(CIFT,INF)->@CIF, DUP(0)->BCAP,->@LNO,
                ELSE OUTCIM,CIF NE 0
                  THEN <WHILE DUP(INCH(CIF)) NE EOFF START
                  OPCH(<=>COF),REPEAT,LOSE,
                  CLOSEF(CIF),DUP(0)->@CIF,DUP->@LNO,DUP->@BFR,->@CHGS> >>
       WHILE DUP NE LNO START
           OUTCIM, RDNIM(1) EQ 0 THEN < LOSE, EXIT > REPEAT
       LOSE,
       VERI NE 0 THEN <PRINT>
       EXIT
.
. common function for F and L.
FN BLDSTR: ENTRY
       MODFS,
       LBFR NE 0 THEN<CURCOL, BLANKS, EOSTR THEN<->@CURCOL,
            ELSE LOSE, EXIT >>
       INSTRING(LSTR,@LSTR,@EOSTR)->@LSTR,
       EXIT
.
. find directive.
REF F: ENTRY
       BLDSTR,
       OUTRDN EQ 0 THEN<EXIT>
       WHILE MATCH(LSTR,@BCAP) EQ 0 START
           OUTRDN EQ 0 THEN<EXIT>
           REPEAT
       VERI NE 0 THEN<PRINT>
       EXIT
.
. locate directive.
REF L: ENTRY
       BLDSTR,
       OUTRDN EQ 0 THEN <EXIT>
       WHILE MATCH(LSTR,@BCAP) EQ 0 START
           LBFR LE (BFR-BCNO) THEN<ADV(@BCNO),
               ELSE OUTRDN EQ 0 THEN<EXIT>>
           REPEAT
       VERI NE 0 THEN <PRINT>
       EXIT
.
. data and functions for the change directive.
VAR DELCHR:0   . delimiter char.
VAR LSTRA:<>, VAR LBFRA:0, RESERVE 39,
VAR BCAPA:<
VAR BCNOA:0,>
VAR BFRA: RESERVE 68,
.
. function to read in first substring for change.
FN BLDSSTR:ENTRY,
       LSTRA,INSTRING(@LSTRA,[CHAR NE DELCHR])->@LSTRA,
       EXIT
.
. function to try to match substring for change.
FN LMATCH: ENTRY,
```

```
        WHILE  MATCH(LSTRA,@BCAP) EQ 0 START
            LBFRA GT (BFR-BCNO) THEN <0,EXIT>
            ADV(@BCNO), REPEAT,
        BCNO-LBFRA->@BCNO, 1,
        EXIT
.
. c - change directive.
REF C: ENTRY,
        MODFS,
        STCOL-1->@BCNO,
        CHAR->@DELCHR, ADVCH,
        BLDSSTR, LMATCH, ADVCH,
        0,DUP->@BCNOA, ->BCAPA,
        THEN <BCNO+LBFRA+1 LT BFR THEN
                <BCNO, DUP+LBFRA->@BCNO,
                @BCAPA,
                WHILE DUP, GETCH(@BCAP), DUP NE 13 START
                    PUTCH(,,), ADV(@BCNO),
                    ADV(@BCNOA), REPEAT
                BCNOA->BCAPA,
                LOSE,LOSE,LOSE,->@BCNO>
            @BCAP,
            WHILE CHAR NE DELCHR START
                 DUP, CHAR, PUTCH(,,),
                 ADV(@BCNO), ADVCH,
                 REPEAT,
            0->@BCNOA,
            WHILE DUP, GETCH(@BCAPA),DUP NE 0 START
                 PUTCH(,,), ADV(@BCNO),
                 ADV(@BCNOA),
                 REPEAT,
            LOSE, LOSE, LOSE,
            BCNO+1->@BFR, PUTCH(@BCAP,13), CHG,
            VERI NE 0 THEN <PRINT>
        ELSE
            OUTST(' No match. '), OPNL,
            WHILE CHAR NE DELCHR START ADVCH REPEAT>
        ADVCH,
        EXIT
        ENDBLOCK
DIR SVBLK:ENTRY, SAVBLOCK(@EDDIR), EXIT .
     SVBLK           .
```

12.7 Instruction Execution Analysis

This example provides a general tool for the analysis of any MINT text execution. The instruction analysis routines use the TRAP operator to gain control at the start of execution of each MINT instruction. Thus, it is possible to tabulate the usage of each instruction and function reference within the text which is under analysis. The Section below lists the required source text. This text includes commentary to explain the use of the trace functions. After this listing, an example use is shown.

12.7.1 Listing of Instruction Trace Functions

```
.   Trace module to obtain VM instruction mix
.
.     Call: TRMIX (<start>)
.           <start> is address of ENTRY for path to be traced until EXIT.
.           Report is printed at EXIT.
.     Call: TREND
.           Will terminate TRMIX tracing.
.
FN TRMIX                        . Take mix over function path
FN TREND                        . End TRMIX
.
    UNLOCK INTDIC
    BLOCK
.
LAB TRMFCNA
LAB TRMFCNB
VAR TRMFR:0
VAR CNTRS: RESERVE 80
VAR STRT :0, VAR ENDLCN:0,
VAR SCR  :0, VAR TOTAL :0, VAR SUM :0,
VAR IGFLG:0, VAR KLIST :0, VAR SCR2:0,
VAR FCNBUF:0,6,VAR FCNCHN:0, VAR FCNLIST:0, VAR CURFCN:0,
VAR OP0:'Ill', VAR OP1:'GET', VAR OP2:'GETV',VAR OP3:'VAL', VAR OP4:'->',
VAR OP5:'DUP', VAR OP6:'LOSE',VAR OP7:'GLKP',VAR OP8:'SLKP',VAR OP9:'+',
VAR OP10:'-',  VAR OP11:'*',  VAR OP12:'/',  VAR OP13:'NEG',
VAR OP14:'FROM',VAR OP15:'MASK',VAR OP16:'UNION',VAR OP17:'DIFFER',
VAR OP18:'COMPL',VAR OP19:'EQ',VAR OP20:'NE',VAR OP21:'LT',VAR OP22:'GT',
VAR OP23:'LE', VAR OP24:'GE',VAR OP25:'NOT',VAR OP26:'AND',VAR OP27:'OR',
VAR OP28:'XOR',VAR OP29:'CHOOSE',VAR OP30:'YES',VAR OP31:'NO',
VAR OP32:'GO',VAR OP33:'DO', VAR OP34:'ENTRY',VAR OP35:'EXIT',
VAR OP36:'GETCH',VAR OP37:'PUTCH',VAR OP38:'INCH',VAR OP39:'OPCH',
VAR OP40:'MATCH',VAR OP41:'DICMATCH',VAR OP42:'STOP',VAR OP43:'PDUMP',
VAR OP44:'TIME',VAR OP45:'OPENF',VAR OP47:'CLOSEF',VAR OP48:'EXR',
VAR OP49:'TRAP',VAR OP50:'<=>', VAR OP51:'TRUE',VAR OP52:'FALSE',
VAR OP53:'ADV',VAR OP54:'ESTOP',VAR OP55:'ADIFF',VAR OP56:'-->',
VAR OP57:'<--',VAR OP58:'VMDEBUG',VAR OP80:'EMULATE',
VAR OPTBL:0,OP1,OP2,OP3,OP4,OP5,OP6,OP7,OP8,OP9,OP10,OP11,OP12,OP13,OP14,
      OP15,OP16,OP17,OP18,OP19,OP20,OP21,OP22,OP23,OP24,OP25,OP26,OP27,
      OP28,OP29,OP30,OP31,OP32,OP33,OP34,OP35,OP36,OP37,OP38,OP39,OP40,
      OP41,OP42,OP43,OP44,OP45,0,OP47,OP48,OP49,OP50,OP51,OP52,OP53,OP54,
      OP55,OP56,OP57,OP58,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,OP80,
.
FN  TRMFNR: ENTRY, GO TRMFR, EXIT
.
.   Space
.
FN SPC: ENTRY, WHILE DUP NE 0 START OB, -1, REPEAT LOSE EXIT
.
.   Calculate and print percent value
.
FN CALC%: ENTRY, ->@SUM,
    SETOPP(#0,2) OB DUP(*100) OPINT(DUP(/SUM)) OPCH(SOUNIT,#.)
    OPINTD((*SUM,-)*100/SUM) SETOPP(# ,5)
```

```
    EXIT
.
.   Activate trace
.
FN MIXCOM: ENTRY,
    0, WHILE DUP NE 79 START DUP, 0->(<=>, FROM @CNTRS), +1, REPEAT LOSE
    OUTST('Instruction mix trace activated.') OPNL TRAP(@TRMFNR),
    EXIT
.
.   Trace function common for detailed analysis
.
LAB FCNCOM:
    WHILE IGFLG NE 0 THEN <DUP EQ VAL(1 FROM IGFLG), FALSE TRAP
        POPUP(@IGFLG),GO BACK>
    ADV(@TOTAL),
    VAL(DUP) GE 80 THEN <@KLIST
        WHILE DUP(VAL) NE 0 START, ->@SCR, VAL(DUP) EQ VAL(1 FROM SCR)
            THEN <PUSHD(DUP+1, @IGFLG), VAL(2 FROM SCR),
                ELSE SCR, REPEAT>
            ELSE VAL(DUP)>
        FROM @CNTRS->@SCR,
    VAL(SCR)+1->SCR,
.
    VAL(DUP) EQ @ENTRY THEN<@FCNCHN,
         WHILE DUP(VAL) NE 0, LAB z , NO z , ->@SCR,
             DUP NE VAL(1 FROM SCR) THEN<
                 SCR, GO BACK, z:, FORGET z, NEXTFREE(@FCNBUF)->@SCR,
                 JOIN(SCR, @FCNCHN),
                 DUP->2 FROM SCR,DUP->3 FROM SCR,->4 FROM SCR,
                 DUP->1 FROM SCR>
             VAL(4 FROM SCR)+1->4 FROM SCR, GLKP->5 FROM SCR,
             PUSHD(CURFCN, @FCNLIST), SCR->@CURFCN>
LAB COMA: CURFCN NE 0 THEN <GLKP EQ VAL(5 FROM CURFCN)
             THEN <VAL(2 FROM CURFCN)+1->2 FROM CURFCN, TRAP>
        POPPEDUP(@FCNLIST)->@CURFCN, GO COMA>
    TRAP,
.
.   Print Mix report
.
FN REPORT: ENTRY OPNL
    TOTAL EQ 0 THEN<OUTST('No instructions traced.'), OPNL, EXIT>
    OUTST('Instructions traced: ') OPINT(TOTAL) OPNL OPNL
    OUTST('    Op. Code       Count    % ') OPNL
    OUTST('    FCN"s         ') SETOPP(# , 4) OPINT(DUP(CNTRS))
    CALC%(TOTAL) OPNL, 1,
    WHILE DUP NE 80 START
        DUP FROM @CNTRS->@SCR,
        VAL(SCR) NE 0 THEN<SETOPP(# , 5), OPINT(DUP), SPC(4),
            VAL(DUP FROM @OPTBL) NE 0 THEN<OUTST(VAL(DUP FROM @OPTBL)),
                VAL(VAL(DUP FROM @OPTBL)), WHILE DUP NE 8 START
                   OB, +1, REPEAT, LOSE>, SETOPP(# , 4),
            OPINT(DUP(VAL(SCR))), CALC%(TOTAL), OPNL>
        +1,
    REPEAT, LOSE,
    DUP(0)->@TOTAL, WHILE DUP NE 79 START
        DUP, 0->(<=>, FROM @CNTRS), +1,
    REPEAT, LOSE,
```

```
    EXIT
.
.   Detailed fcn report
.
FN REPORTF: ENTRY,
    WHILE FCNLIST NE 0 START POPUP(@FCNLIST) REPEAT
    FCNCHN EQ 0 THEN <EXIT>
    OPNL OUTST('Functions called:') OPNL OPNL
    OUTST('Address Entry    Number  Count    %     ') OPNL,
    CUR
    WHILE FCNCHN NE 0 START
        JOIN(DETACHED(@FCNCHN), @FCNBUF),
        OPINT(DUP(VAL(1 FROM FCNBUF))), OB, LOOKD,
        SETOPP(# , 6), OPINT(VAL(4 FROM FCNBUF)),
        SETOPP(# , 6), OPINT(DUP(VAL(2 FROM FCNBUF))), CALC%(TOTAL),
        OPNL,
     REPEAT, ->@CUR, 0->@CURFCN,
    EXIT
PAGE
.   Initiate Tracing
.
TRMIX: ENTRY, ->@STRT, @TRMFCNA->@TRMFR, MIXCOM, EXIT
.
.   Discontinue tracing
.
TREND: ENTRY, TRAP(0), REPORT, EXIT
.
TRMFCNA: DUP EQ STRT
    THEN <GLKP->@ENDLCN, @TRMFCNB->@TRMFR, GO FCNCOM>
    TRAP,
.
TRMFCNB: VAL(DUP) EQ @EXIT AND GLKP EQ ENDLCN THEN<
        OUTST('Instruction mix for path from '), OPINT(STRT),
        OUTST(' to '), OPINT(DUP),
        TOTAL, REPORT, ->@TOTAL, REPORTF, 0->@TOTAL,
        @TRMFCNA->@TRMFR, TRAP>
    GO FCNCOM,
.
ENDBLOCK
    RCL$
PAGE
```

12.7.2 Example of TRMIX Use

The following is an example use of the instruction analysis routine. The routine which is analyzed is the insert directive from the edit routines.

```
MINT-3 Virtual Machine (32-bit Virtual Memory): Version 1.1
VSTORE size 16777K words.  Start PDUMP load......:
MINT-3 System: Version 3.0.  Created on: 020216
Copyright D.F. Hendry, 1990
VM>SI utils/tracemx.mnt
VM>SI utils/edit.mnt
VM>EDIT NEWFILE
```

```
MINT Text Editor 1.5. Date: 020218
VM>NOW TRMIX(@I)!
Instruction mix trace activated.
VM>I this is a test line of text.
Instruction mix for path from  41355 to  41360
Instructions traced:  03075

    Op. Code       Count    %
    FCN"s            95  03.08
     1    GET       874  28.42
     2    GETV      390  12.68
     3    VAL       266  08.65
     4    ->        239  07.77
     5    DUP       121  03.93
     6    LOSE        2  00.06
     9    +          88  02.86
    14    FROM      147  04.78
    15    MASK      147  04.78
    16    UNION      29  00.94
    19    EQ          4  00.13
    20    NE         31  01.00
    22    GT         31  01.00
    31    NO         66  02.14
    32    GO         59  01.91
    34    ENTRY      96  03.12
    35    EXIT       95  03.08
    50    <=>        58  01.88
    53    ADV        60  01.95
    56    -->        89  02.89
    57    <--        88  02.86

Functions called:

Address Entry    Number  Count    %
 40800                1      5  00.16
 34210 PUTCH         29   1160  37.72
 40905                1    610  19.83
 40833                1      7  00.22
 40881                1      8  00.26
 34669 ADVCH         30    120  03.90
 34154 GETCH         30   1140  37.07
 34674 CHAR           1      9  00.29
 40813                1     11  00.35
 41355 I              1      5  00.16
VM>
```

13. The VM(M) Virtual Machine

13.1 Introduction

This Chapter defines the architecture of the VM(M) Virtual Machine and defines its instruction set. Emphasis is placed on the definition required for implementation by interpretation or by micro-coding.

13.2 The Virtual Machine Architecture

The Virtual Machine consists of 4 major components:

- Main storage
- 2 Stacks
- Instruction execution unit
- Instruction set.

These components are defined in detail below. This definition is sufficient for practical construction of a VM(M) machine.

13.2.1 Main Storage

The Virtual Machine main memory consists of two blocks of contiguous storage units. The first block, used for data storage, must have storage units of at least 32 bits. The second block, used for program storage, should also have storage units of at least 32 bits. Earlier versions of MINT allowed data storage of 16 bits and program storage of a minimum of 8 bits. The system could still be compiled using such values, but this is no longer likely to be useful. This addressable space is referred to as Virtual Store or VSTORE. The main storage may be any size in a particular implementation, bearing in mind that the MINT compiler together with the M-TRAN syntax analysis system occupy about 6000 storage units. All instructions and data are held in main storage. Section 1.10 and Figure 1-2 provide more detail about storage organization.

13.2.1.1 Fixed Address Assignments

The first 81 locations in VSTORE cannot be used for normal programming since their addresses overlap with the values assigned to the VM(M) operation codes. Therefore, this area is used to hold various system data items. The current fixed VSTORE definitions are given in Table 13-1.

Table 13-1 VSTORE Fixed Address Assignments

Location	identifier	description	
0		0	
1	MAXDS$	no. of 128 word blocks in data-space	
2	MAXPS$	no. of 128 word blocks in procedure-space	
3	IDLOC$	initial DLOC value	
4	CONT$	address of abnormal status routine	
5 L	SYSDAT$	6	
6		yy mm	date of system
7		dd	generation
8	MAXVS$	no. of 1024 word blocks in VSTORE	
9	EXOPT$	VM execution options	
10		word 2 of EXOPT$	
11	DREM	divide remainder	
12 L	DATE	6	
13		yy mm	current date
14		dd	
:			:
21 L	SYSID$	length of ID in characters	
22		system ID	
:			:
:			:
:			:

Locations 8 through 10 and 13 through 14 are set by the interpreter when VSTORE is loaded. The variable MAXDS$ is computed by the AUTO directive and is $N_d/128$ (See Figure 1-2 and Section 14.6.1.1.). Similarly, MAXPS$ is $N_p/128$. The variable IDLOC$ is set to the initial value required for DLOC. Initialization text in the compiler stores this value in DLOC. The VM sets the value of MAXVS$ to the value, in 1024 word blocks, of the top of Virtual Memory. The two word EXOPT$ field is used to contain information concerning execution options, and for Virtual Machine implementation indicators. This field is taken to be formatted as 26 bits for option letters a through z and 6 bits for indicators 1 through 6.

The format is as shown in Table 13.2.

Table 13-2 EXOPT$ Definition

word	meaning
9	a - p
10	q - z, 6 - 1

The identifiers DATE, SYSDAT$, and SYSID$ are introduced in class LAB and, therefore, are not modifiable. All other locations are initially set in VSTORE and may be reset during execution.

13.3 Virtual Machine Object Text Format

The MINT compiler and any systems written in MINT are distributed in a standard format. This format, termed *portable format,* is composed of text images containing blanks, hexadecimal numbers, and the characters comma(,), slash (/), minus(-), and period(.). Each text image is 80 characters or less, and is terminated by a checksum and sequence number. A blank character (040) follows the sequence number as the last character of the image. As there are no embedded blanks, this may be used as an end-of-image indicator. In all cases except one, numbers whose value is zero are suppressed. Thus, there may be sequences such as ,, which are to be interpreted as ,0,. The end-of-load sequence will normally be written without zero suppression. The comma, slash, minus, and period characters act as delimiters and indicators of the meaning of the preceding number. Table 13-3 defines the meaning of the delimiters.

Table 13-3 Portable Format Delimiters

delimiter	meaning of preceeding number
,	unsigned data
/	address
-	signed data
.	checksum

A character whose value is 023 indicates end-of-file. This is only used in situations where an entire block of text must be split into more than one

physical file. A zero address value indicates the end of the load process.

The checksum is the accumulated XOR of each number in the image up to the checksum number. The checksum is computed in the portable-format output text, and should be checked in all portable-format read procedures. The image sequence number starts at one and is incremented by one for each successive image. Correct incrementation should also be checked on reading. Note that the checksum and sequence numbers are, like all numbers in portable format, in hexadecimal.

13.4 Loading the Virtual Machine

The Virtual Machine is loaded from a text file based on the field definitions given in the previous Section. The first stage of loading is to set the contents of all Virtual Store locations to zero. Next, the data values from the object text file are loaded into sequential store locations starting with location 0. When an address is encountered (delimiter character is slash) subsequent data values are loaded into VSTORE starting at that address. The load process continues until an address specification of zero is encountered. This signifies the end of the object text file. Next, the interpreter supplied values in low VSTORE (locations 9-10 and 13-15) should be set. At this point the loaded program may be executed. This is always effected by storing the start of procedure-space in the top of the link stack, and starting the Virtual Machine.

Portable-format object text may be recorded on any medium which is acceptable to the target machine. If the medium is not line image oriented, such as paper tape, then the fields may simply continue serially until the 0/ field. In some environments it may be necessary to take steps to deal with C/R-L/F sequences or other end-of-image indicators.

13.4.1 Example Portable Format Loading

The operation of the portable-format load process may be shown by examination of a sample of portable-format text and a procedure for loading portable-format text into VSTORE. The sample text is from the beginning of the diagnostics and is as follows:

```
1/100,17,1C8,8977,6,3238,3730,3831,C/6,10/4,6944,6761,15/1A,2A2A,9088.1
17/2A2A,5320,4154,4B43,4520,4358,5045,4954,4E4F,2A20,2A2A,D2A,2F0C.2
23/25/11,2A20,2A2A,202A,4146,4C49,4445,2A20,2A2A,2A,3,4241,43,2B5C.3
32/2F,,6,25,4D56,4D28,2029,6964,6761,6F6E,7473,6369,2073,6170,171B.4
40/7373,6F20,656E,6320,6D6F,6C70,7465,6465,2E,FFFF-FFFE-FFFD-F490.5
```

After initialization of VSTORE and setting of LP to zero, a portable-format loader must start to accumulate characters and test each one against the set of special characters. When a special character is encountered the defined action must be taken. This is shown by the text listed below. Note that, after the initial references to FNAME and IPAR, this text does not reference any compiler procedures. If the FNAME and IPAR references are replaced by explicit names and values, this routine could be used as an initial VSTORE loader. For this purpose it should be established at a VSTORE address above where loading will take place. At the end of the load this routine executes a GO 32768, which transfers control to the normal initial program start address. Also note that the manipulation for the - control character is appropriate for a loader operating on a ones complement machine. If the loader is to operate on a two's complement machine, - can be treated just like the , character.

13.4.2 Listing of PFLDR

```
.
.   ***   Portable format loader
.
.  ref: PFLDR <file> <data start> <size> <program start> <size>
.
VAR SUM:0,                    . Temp sum of numerical value
VAR CSUM:0,                   . checksum.
VAR SEQ:0,                    . sequence counter.
VAR DSIZE:0, VAR PSIZE:0,     . data and program size.
VAR DS:0, VAR PS:0,           . data and program starts
VAR OFSET:0,                  . current offset
VAR ADD: 0,                   . Next address to be stored into
VAR PFIN:0,                   . Text input segment id
VAR TMP:0,                    . Temp for current character
.
FN STORE: ENTRY,              . Store value
    SUM->(ADD FROM OFSET),
    1 FROM ADD->@ADD,
    EXIT
.
FN ZERO: ENTRY,               . Zero VSTORE
    0->@SUM,
    WHILE DUP GT ADD START STORE REPEAT,LOSE
    EXIT
.
FN DMP:ENTRY, ->@TMP,EXIT
.
FN CKS:ENTRY,                 . Accumulate checksum
    XOR(SUM,CSUM)->@CSUM, EXIT
.
FN ACNUM:ENTRY,
    (TMP GE #0) AND (TMP LE #9) THEN <SUM*16+TMP-#0->@SUM>
    (TMP GE #A) AND (TMP LE #F) THEN <SUM*16+TMP-#A+10->@SUM>
EXIT
```

```
.
FN SEQN:ENTRY,
    SUM NE CSUM THEN<OUTST('Checksum error at: '),OPINT(SEQ),
      OPNL,ESTOP> 0->@SUM,
    WHILE DMP(DUP(INCH(PFIN))) NE 13 START ACNUM REPEAT
    DUP(SEQ+1) NE SUM THEN< OUTST('Sequence error at: '),OPINT,
      OPNL,ESTOP> ->@SEQ, DUP(0)->@CSUM, ->@SUM, EXIT
.
DIR PFLDR: ENTRY,
    OPENF(1,FNAME),IPAR->@DS,IPAR->@DSIZE,IPAR->@PS,
      IPAR->@PSIZE,->@PFIN,32768+PS->@ADD,ZERO(32768+PS+PSIZE),
      DS->@ADD,ZERO(DS+DSIZE),DS->@OFSET, 0->@ADD,0->@SUM,
    WHILE DMP(DUP(INCH(PFIN))) NE 128 START
      TMP EQ #, THEN<CKS,STORE, 0->@SUM>
      TMP EQ #- THEN<CKS,XOR(65535,SUM),+1,NEG,->@SUM,
                     STORE,0->@SUM>
      TMP EQ #/ THEN<CKS,SUM->@ADD,SUM LT 32768 CHOOSE(DS,PS)
                  ->@OFSET, 0->@SUM>
      TMP EQ #. THEN<SEQN>
      ACNUM,
    REPEAT,
    CLOSEF(PFIN),0->@PFIN,
    GO 32768,
PAGE
```

13.4.3 Stacks

The Virtual Machine contains two stacks, each typically 100 storage units in length. Each stack is independently controlled by pointer registers. The two stacks are:

- The operand stack
- The link stack.

All the Virtual Machine instructions operate on operands (parameters) obtained on the stacks. The operand stack is addressed by a register known as the *Stack Pointer* (SP) which increments automatically when an object is obtained on the stack, and decrements automatically when an object is popped from the stack.

The link stack operates in a similar fashion and is addressed by a register called the *Link Pointer* (LP). The link stack is used to control instruction execution sequencing since the instruction unit always executes the instruction pointed to by the top item on the link stack. Normal instruction execution results in incrementation of the top item on the link stack so that the next instruction is addressed. A procedure reference pushes an item onto, and an EXIT pops an item from, the link stack.

13.4.4 The Instruction Execution Unit

The instruction execution unit carries out execution of the Virtual Machine instructions in the following steps:

1. Fetch the instruction addressed by the top item on the link stack,
2. Increment the top item on the link stack by 1,
3. Execute the fetched instruction.

This execution process is continued until an event occurs which causes a Virtual Machine halt.

Instructions are assigned numeric codes. The highest numbered instruction code permitted is 80. If a value greater than 80 is encountered the instruction execution unit performs a procedure reference. The value is processed as the address of a procedure, which causes the procedure address to be pushed onto the link stack. The procedure referencing operations are further explained in Section 13.5.10.

13.5 The Virtual Machine Instruction Set

The following paragraphs describe the instructions in the Virtual Machine. In the diagrams SP and LP always point to the current top of the operand and link stack respectively. The stacks themselves increment down the page. The addresses of the sections of Virtual Store shown increase down the page. Where relevant, store addresses are shown at the upper left corner of the box which represents a unit of storage.

In several cases, contents of sections of the stacks are shown even though the section is below the current stack pointer. The contents of such sections are not logically accessible, but are shown, enclosed in parentheses, for diagrammatic clarity.

13.5.1 Storage Access Primitives

The storage access primitives are operators which obtain objects on the operand stack, store objects in the Virtual Store, or modify the contents of a VSTORE location. The primitives in this group are:

GET
GETV

```
VAL
->
ADV
```

13.5.1.1 The GET Primitive

The GET primitive pushes onto the operand stack the contents of the Virtual Store location following the GET operator, and increments the top item on the link stack by 1. Since the instruction execution unit always increments the top item on the link stack by 1, the effect after instruction completion is that the item has been incremented by 2. The operation of GET is illustrated in Figure 13-1.

13.5.1.2 The GETV Primitive

The GETV primitive is similar to the GET primitive; the difference being that one level of indirect addressing is applied to the operand. The value following the instruction is used as the VSTORE address from which to obtain the item. This is illustrated in Figure 13-2.

13.5.1.3 The VAL Primitive

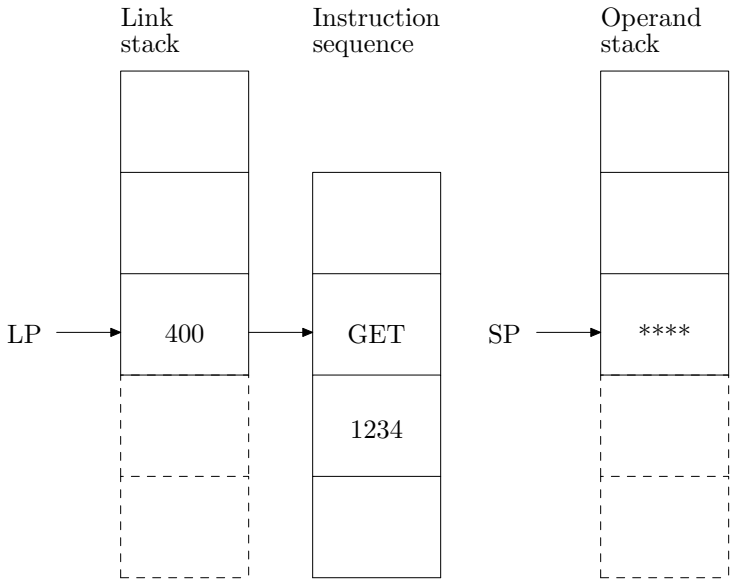
The VAL primitive treats the top object on the stack as a virtual address and replaces it with the contents of that address. See Figure 13-3.

13.5.1.4 The -> Primitive

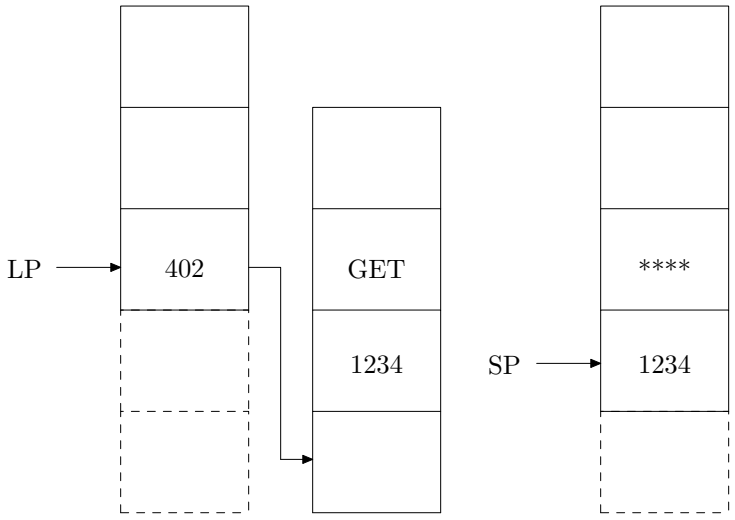
The `->` primitive treats the top object on the stack as a virtual address and stores the next object on the stack in that address. Both these objects are removed from the stack by decrementing the Stack Pointer (SP) two locations. This is illustrated in Figure 13-4. After execution, virtual location 616 contains the value 250.

13.5.1.5 The ADV Primitive

The ADV primitive treats the top item on the operand stack as a VSTORE address, and increments by one the contents of the location at that address. This operation is illustrated in Figure 13-5.

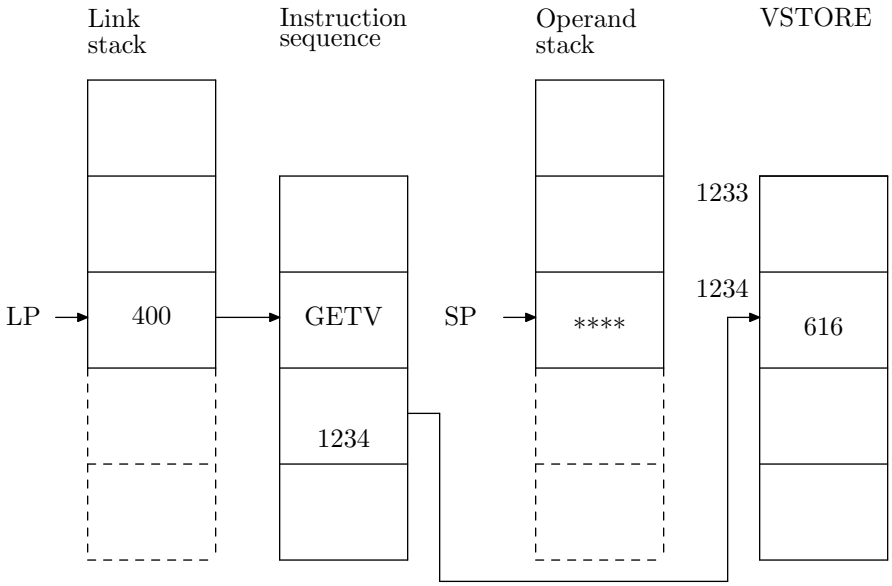


Before Execution

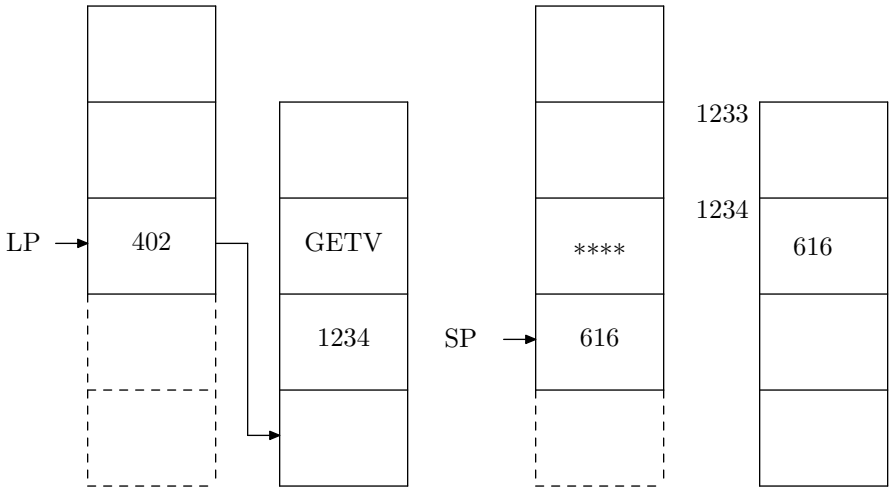


After Execution

Figure 13-1 Operation of GET

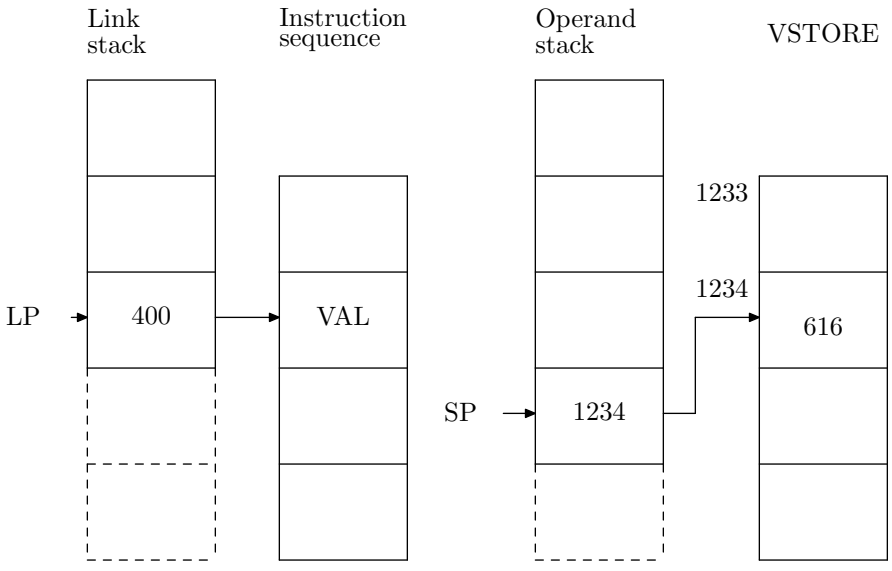


Before execution

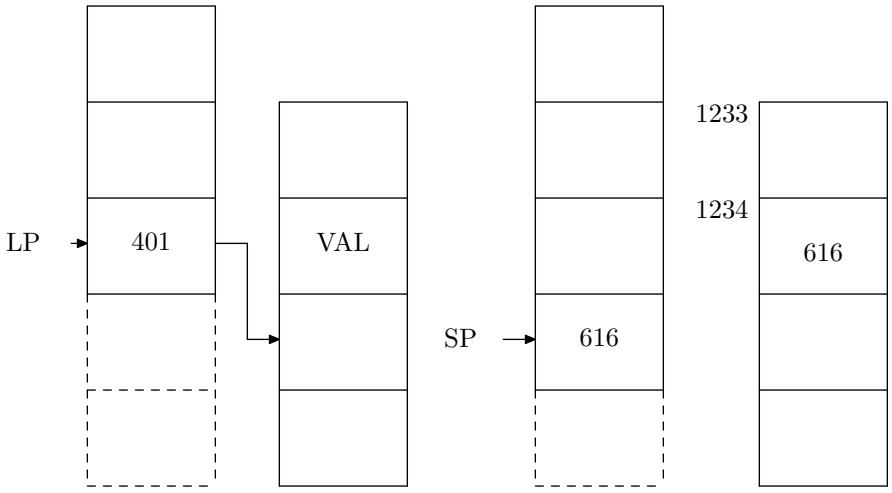


After execution

Figure 13-2 Operation of GETV

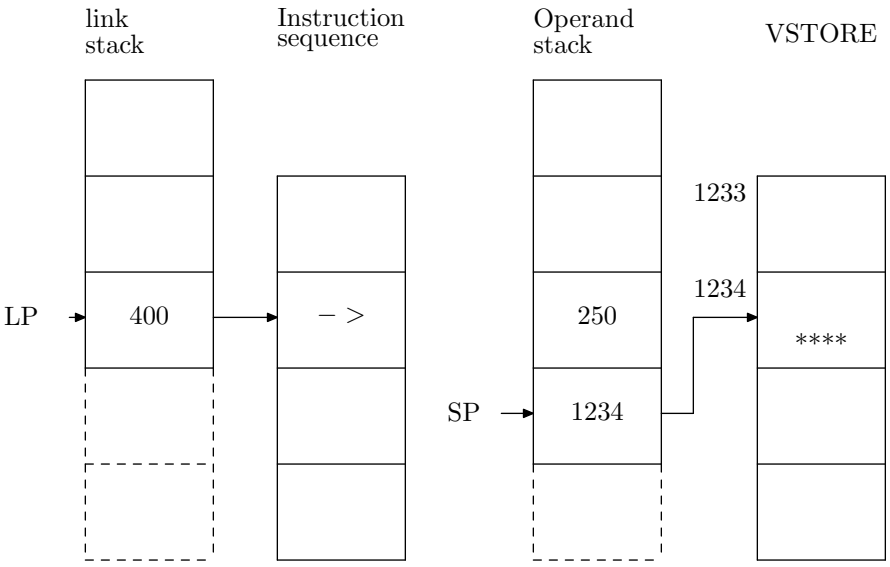


Before execution

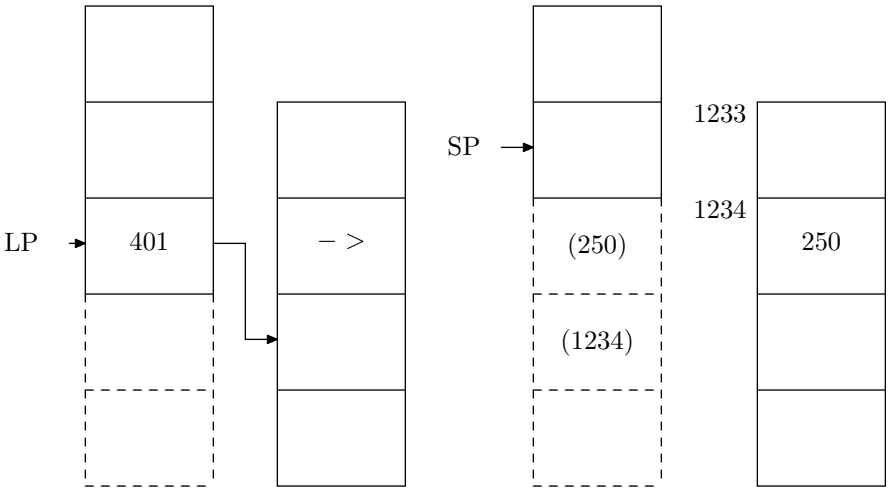


After execution

Figure 13-3 Operation of VAL

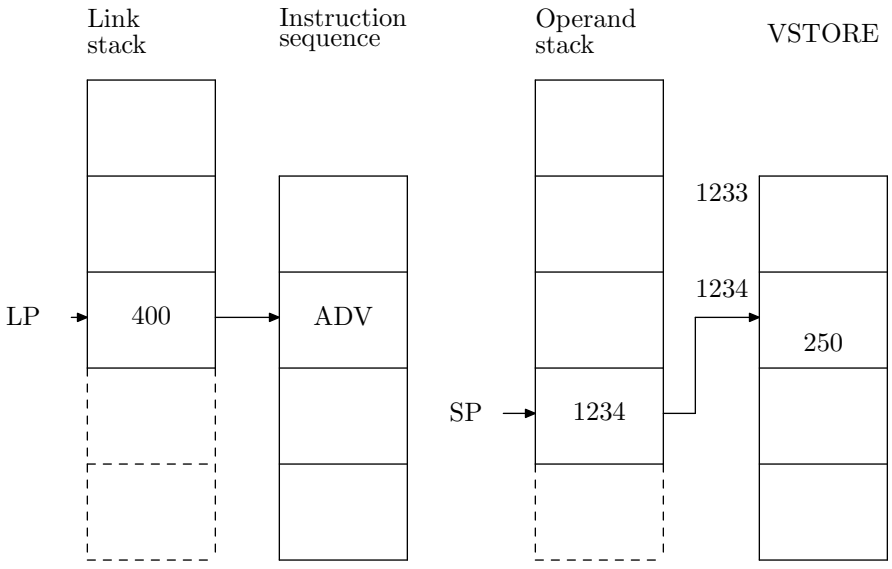


Before execution

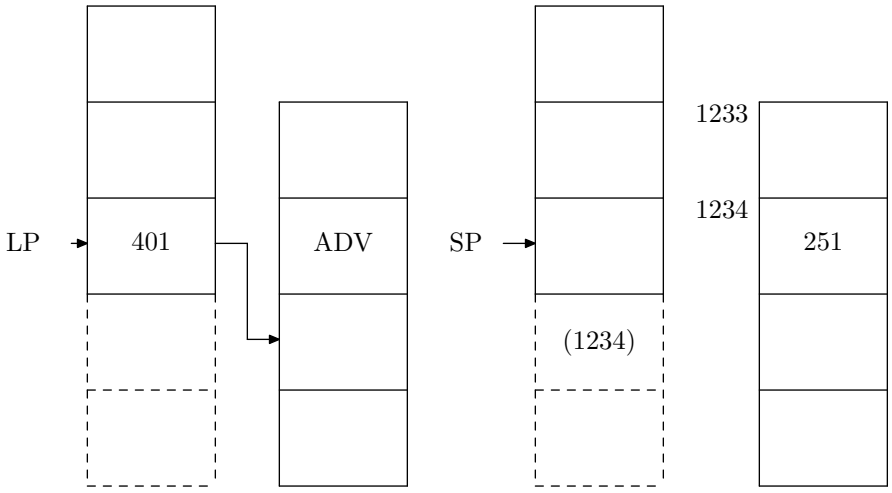


After execution

Figure 13-4 Operation of − >



Before execution



After execution

Figure 13-5 Operation of ADV

13.5.2 Operand Stack Manipulation

The following operators allow efficient manipulation of the operand stack:

DUP
LOSE
<=>

13.5.2.1 The DUP Primitive

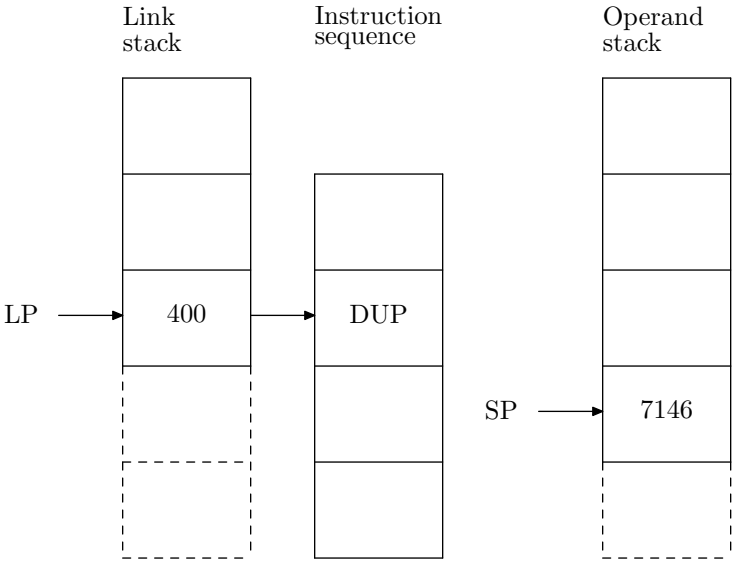
The DUP primitive obtains on the stack a duplicate copy of the top object as illustrated in Figure 13-6.

13.5.2.2 The LOSE Primitive

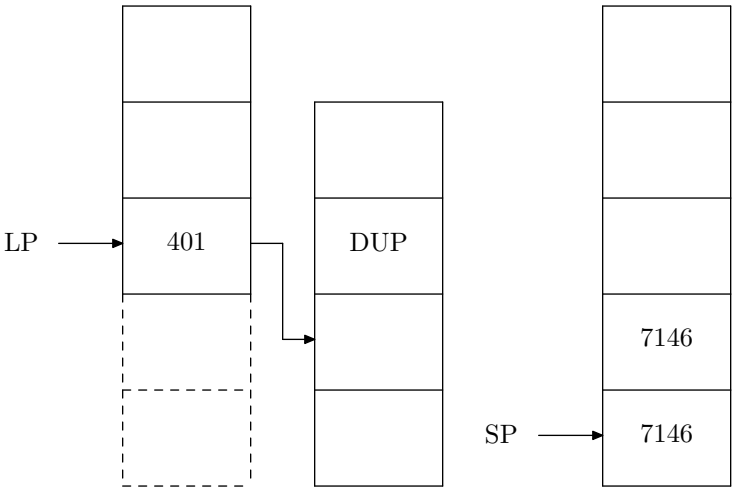
The LOSE primitive discards the top object from the stack as illustrated in Figure 13-7.

13.5.2.3 The <=> Primitive

The <=> primitive removes the top two items on the operand stack. It then first pushes the first object and then the second object. Thus, the order of the two objects on the stack is reversed. The operation of <=> is illustrated in Figure 13-8.

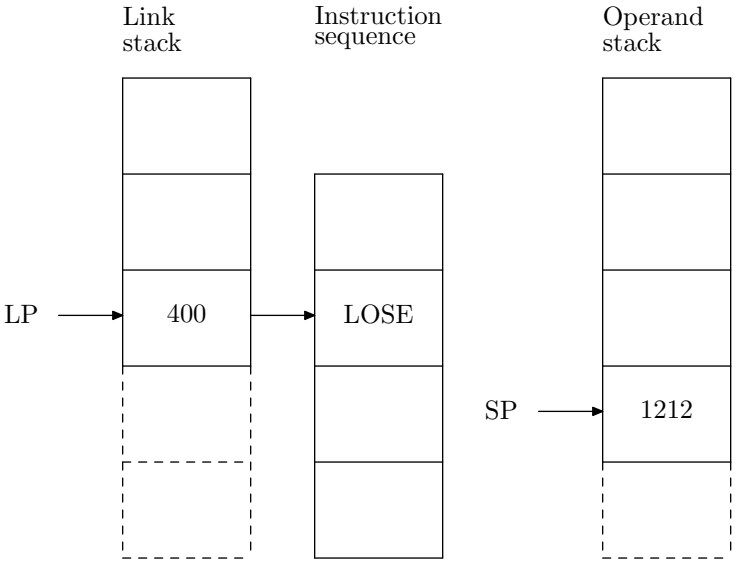


Before execution

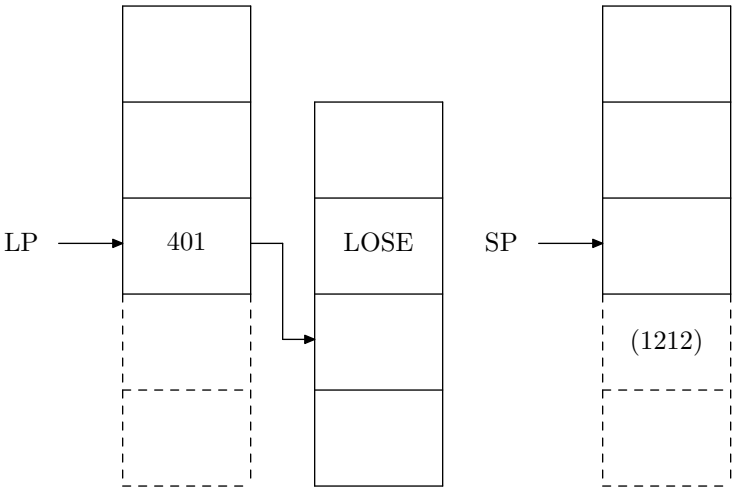


After execution

Figure 13-6 Operation of DUP

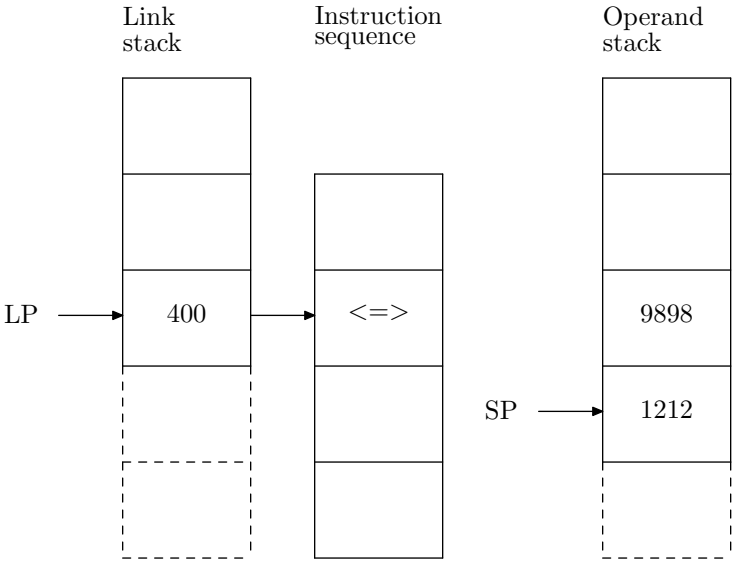


Before execution

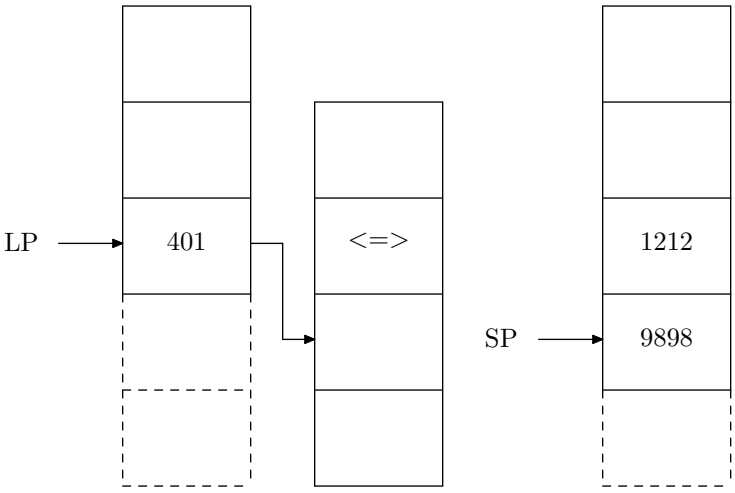


After execution

Figure 13-7 Operation of LOSE



Before execution



After execution

Figure 13-8 Operation of <=>

13.5.3 Link Stack Manipulation

13.5.3.1 The GLKP and SLKP Primitives

The GLKP primitive obtains on the operand stack the current contents of the Link Pointer (LP) register. The address thus obtained is not a valid VSTORE address as neither of the two stacks is located in VSTORE. The only useful operation that may be performed on the obtained LP register contents is to subsequently reload the LP register with it. This is effected by means of the SLKP operator which pops the top object from the operand stack, loads the LP register with it, and then stores the value previously addressed by the LP register at the new LP register address. Figures 13-9 and 13-10 illustrate the operation of GLKP and SLKP respectively.

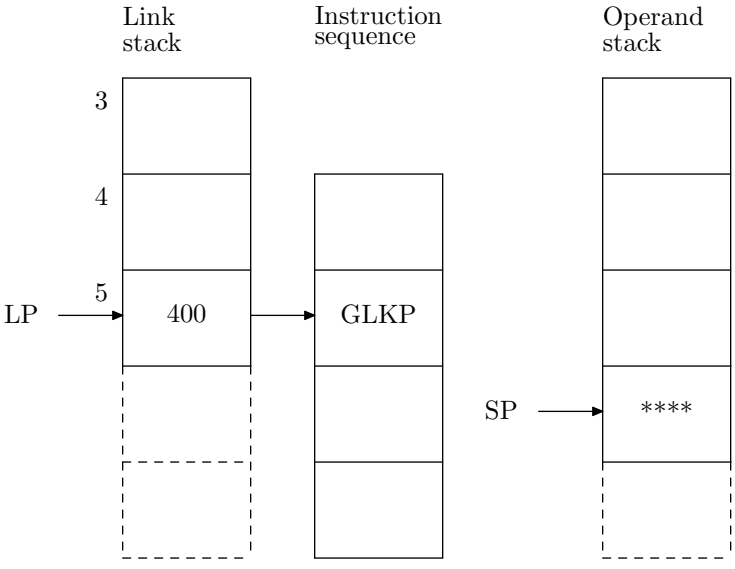
Because of the above restrictions on the use of GLKP and SLKP the instructions are normally used together. On entry to a given function the Link Pointer may be stored in a variable. If this function references other functions it may be desired to exit from the nested function directly to the original referencing text. This is achieved by first setting the Link Pointer from the variable and then executing the EXIT primitive.

Arithmetic operations performed on the accessed Link Pointer are invalid. Thus,

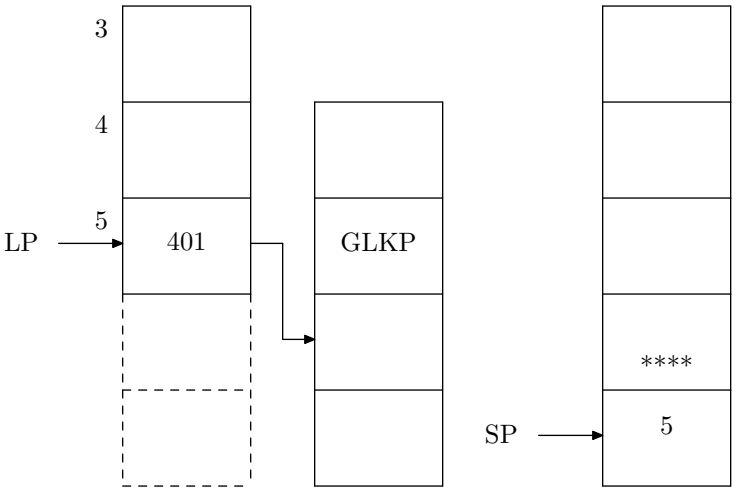
```
SLKP(GLKP-4)
```

must not be interpreted to mean that on execution of the EXIT primitive the user will be exited from a four deep nested function to the original object text reference point.

It is also not valid programming to execute an SLKP instruction with a value on the operand stack which is greater than the current value of LP. The Virtual Machine should treat such an operation as illegal.

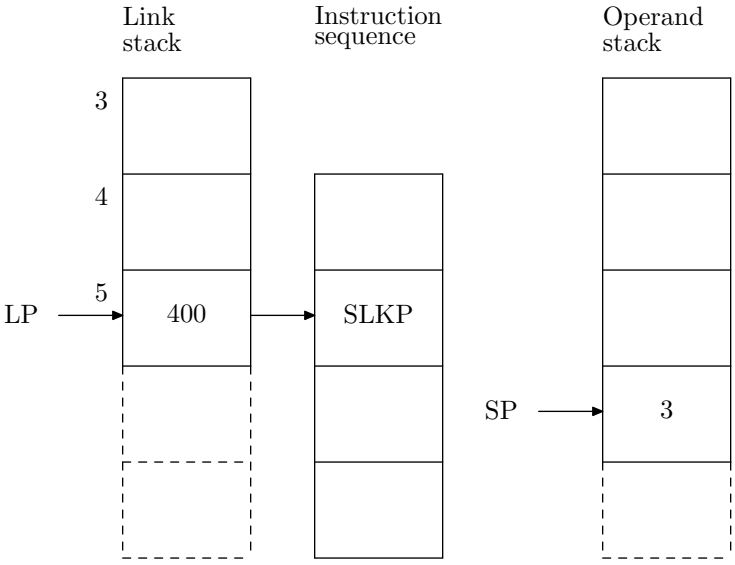


Before execution

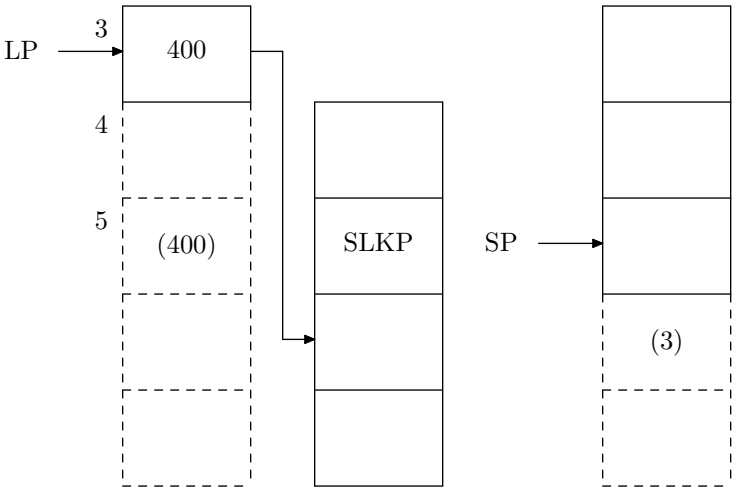


After execution

Figure 13-9 Operation of GLKP



Before execution



After execution

Figure 13-10 Operation of SLKP

13.5.4 Arithmetic Operators

This group of primitives includes the word arithmetic operators and binary operators. In the current Virtual Machine all arithmetic operations are performed on integer quantities only. The operators in this group are:

+
−
*
/
NEG
MASK
UNION
DIFFER
COMPL

Most of the primitives in this group operate on two objects and return a single result on the stack. The exceptions are the NEG and COMPL operators.

At present the result of arithmetic exceptions, such as divide by zero, is left to the implementer of each Virtual Machine. The MINT system itself does not depend on any definition of such results.

13.5.4.1 The + Primitive

The + primitive computes the integer sum of the top two objects on the operand stack and replaces them with a single object which is the result. The + operation is illustrated in Figure 13-11.

13.5.4.2 The − Primitive

The − primitive computes the integer difference between the top two objects on the stack and returns the result to the stack. Note that the order of the operands is significant; the top object on the stack is the second operand, and the next object down is the first operand. This is illustrated in Figure 13-12 by computing 99-71.

13.5.4.3 The * Primitive

The * primitive computes the integer product of the two top objects on the stack and returns it as a single result. The operation logic is the same

as that for + as shown in Figure 13-11, with ∗ substituted for +.

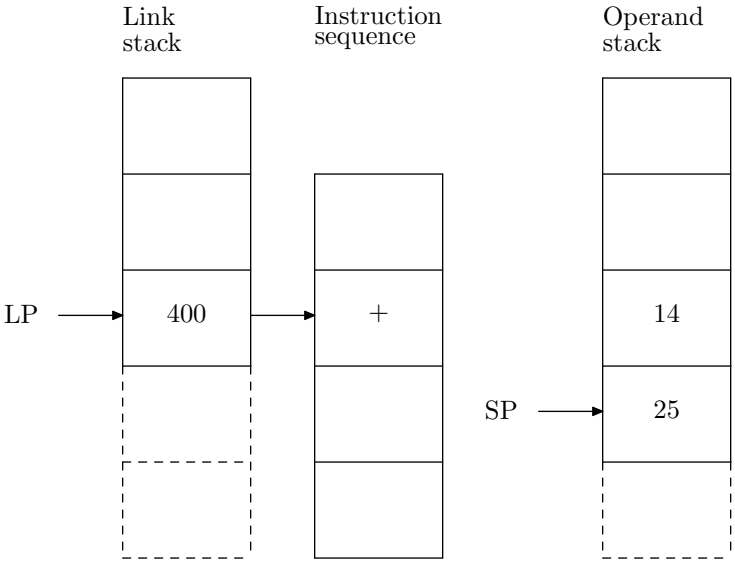
13.5.4.4 The / Primitive

The / primitive computes the integer quotient of the top two objects on the stack and returns it as a single object. The operation logic is the same as that for − as shown in Figure 13-12, with / substituted for −. The remainder is stored in the variable DREM. The quotient is not rounded. Thus,

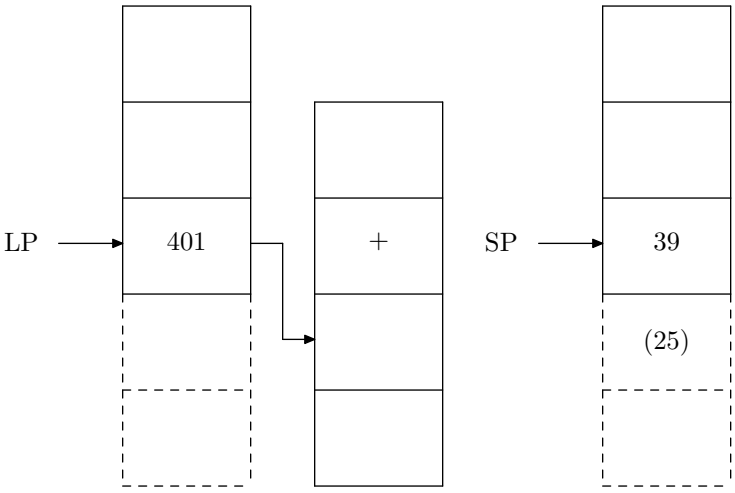
17 / 3 yields 5.

13.5.4.5 The NEG Primitive

The NEG primitive replaces the top operand on the stack with its negative value. This operation should yield the same result as multiplication by minus one. The SP is not modified. See Figure 13-13.

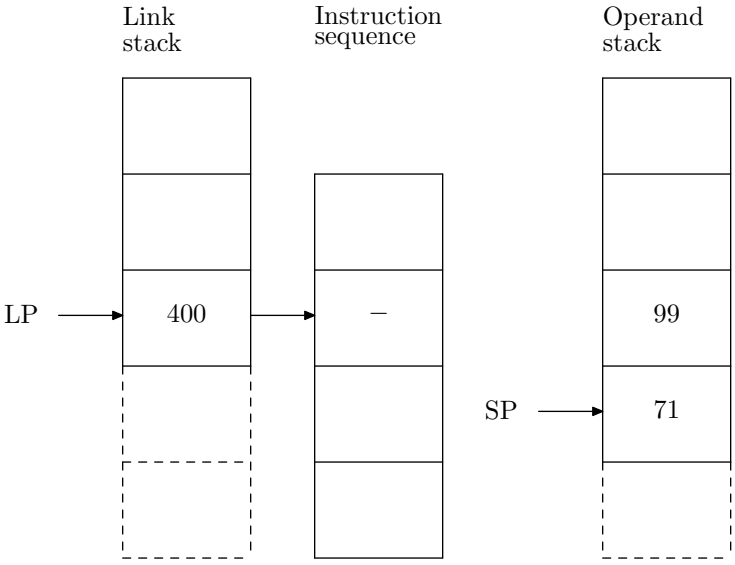


Before execution

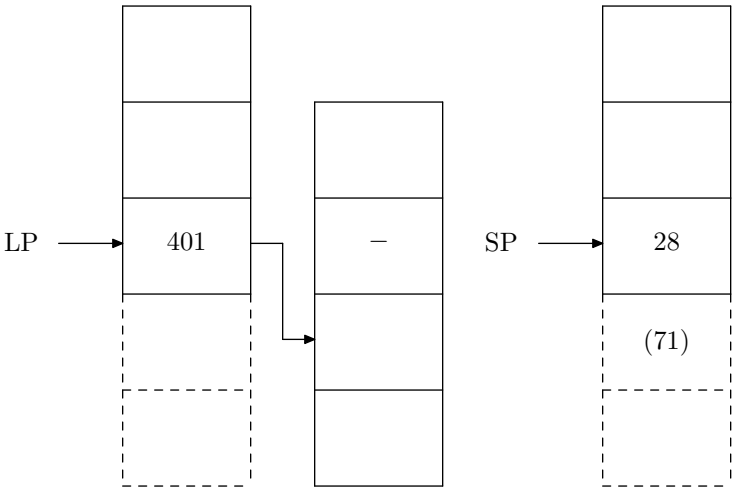


After execution

Figure 13-11 Operation of +

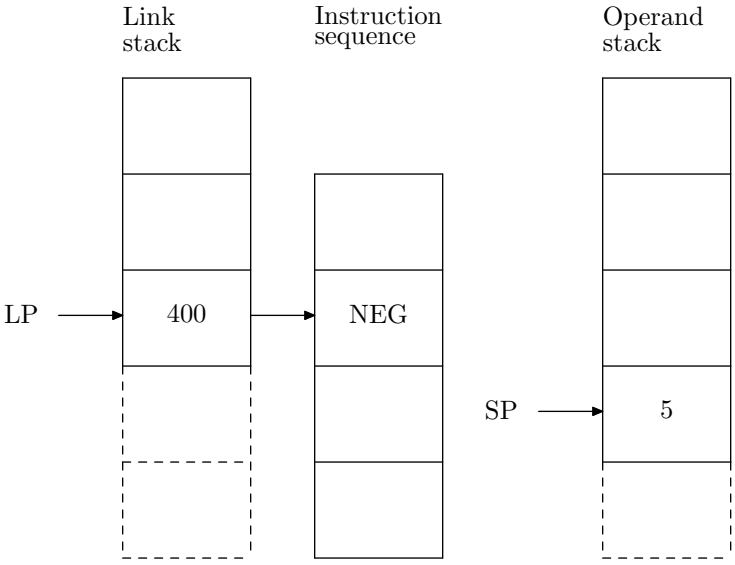


Before execution

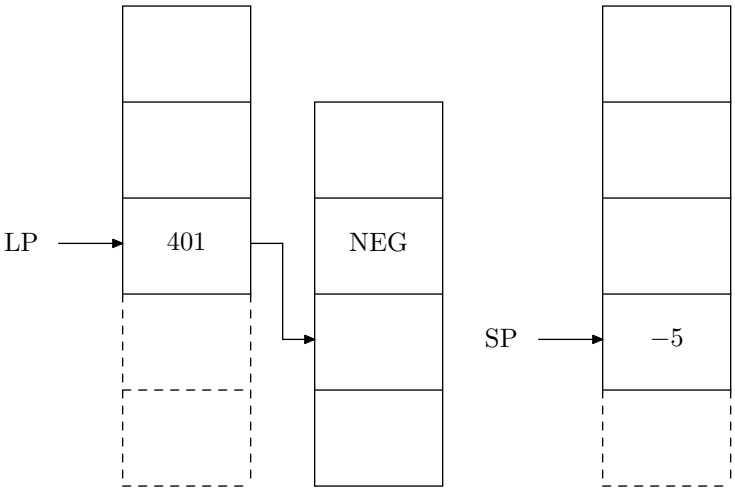


After execution

Figure 13-12 Operation of –



Before execution



After execution

Figure 13-13 Operation of NEG

13.5.4.6 The MASK Primitive

The MASK primitive is a binary arithmetic operator which applies a binary AND to the top two objects on the stack and returns a single result. This is illustrated in Figure 13-14.

13.5.4.7 The UNION Primitive

The UNION primitive applies a binary inclusive or to the top two items on the stack and returns a single result. Thus,

101100 UNION 111001

yields 111101.

13.5.4.8 The DIFFER Primitive

The DIFFER primitive applies a binary exclusive or to the top two items on the stack and returns a single result. Thus,

110011 DIFFER 101100

yields 011111.

The operation Figure for MASK (Figure 13-14) applies for the logic of DIFFER.

13.5.4.9 The COMPL Primitive

The COMPL primitive is analogous to the NEG (Figure 13-13) operator but COMPL replaces the top item on the stack with its binary ones complement. Thus,

COMPL (110101)

yields 001010.

The operation Figure for NEG applies for the logic of COMPL.

13.5.5 Address Arithmetic Operators

13.5.5.1 The FROM Primitive

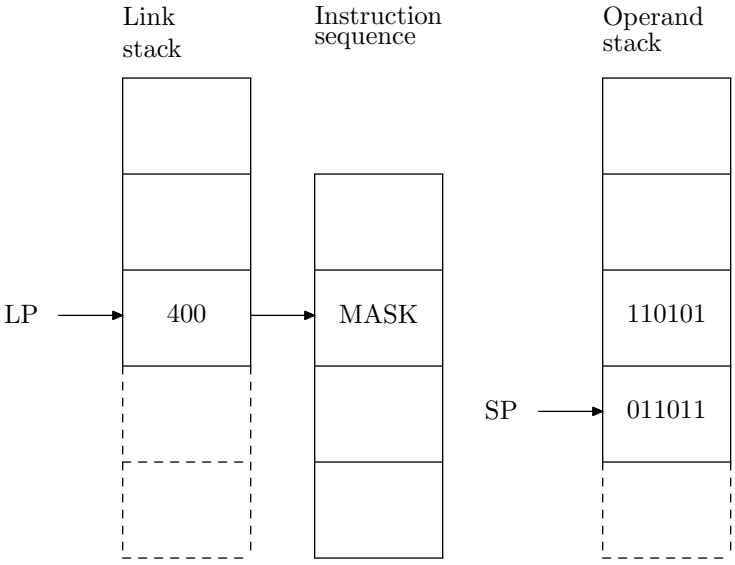
The FROM primitive provides a means of computing offsets from addresses. FROM expects two operands on the stack. It adds these two operands, using address arithmetic, and leaves the result on the stack. Thus, the logic of the operation of FROM is the same as for +. However, the numerical result may not be the same due to possible variation in VSTORE addressing techniques used by Virtual Machines. Normal arithmetic must use +, while all address arithmetic must use FROM. If an implementation makes direct use of byte addressing in the Virtual Machine implementation, and a MINT word is composed of two bytes,

```
5 FROM 100
```

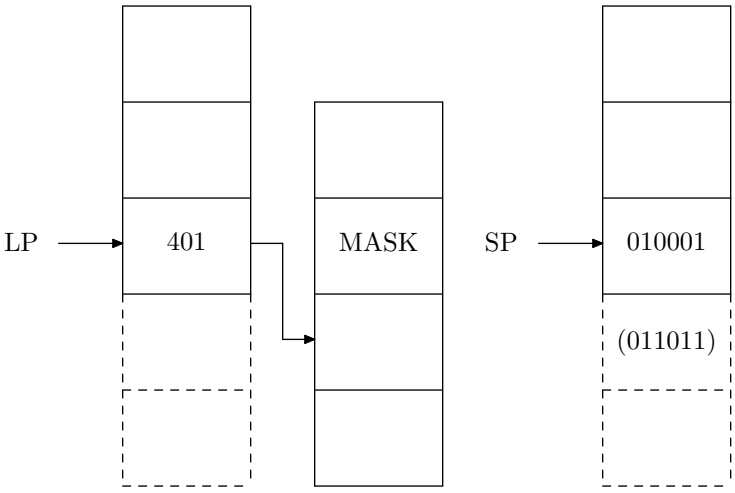
would yield 110. The logic of FROM is as shown for + in Figure 13-11. However, the result must be correct for VSTORE word addresses as explained above.

13.5.5.2 The ADIFF Primitive

The ADIFF primitive computes the difference between two address items. As is true for the FROM operator, the arithmetic must be carried out in the units of addressing in the Virtual Machine implementation. The logic of the ADIFF operator is as shown for the − operator in Figure 13-12.



Before execution



After execution

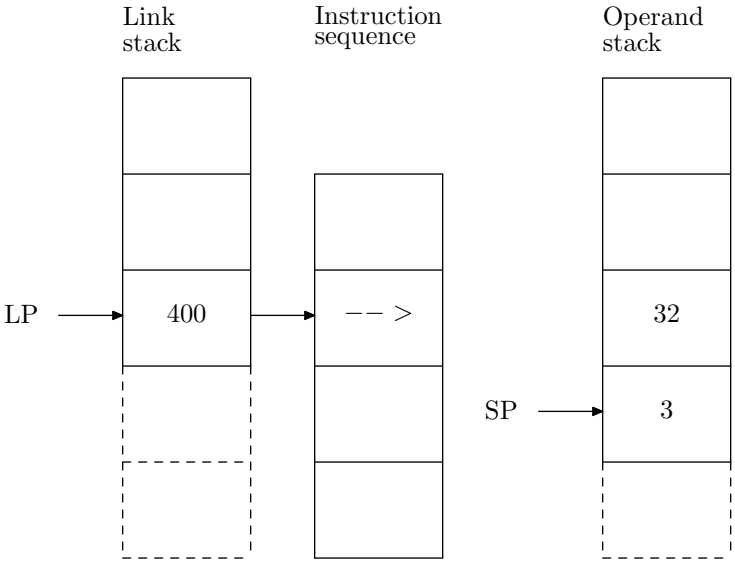
Figure 13-14 Operation of MASK

13.5.6 Logical Shift Operators

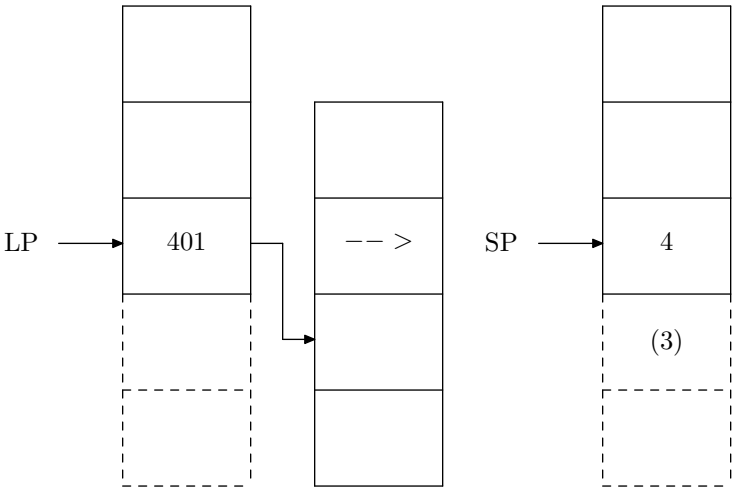
The two operators in this group provide the ability to shift the contents of a word by a number of bit positions from 0 through 31. The operators shift right and left respectively:

`-->`	- right shift logical,
`<--`	- left shift logical.

These are defined as logical shifts and therefore any bits which are shifted out of the word are discarded. For implementations on machines which do not naturally perform operations on 32 bit words it is necessary to mask off any excess bits or otherwise compose the correct 32 bit result. Figure 13-15 shows the operation of `-->`. The operation of `<--` is exactly the same except the data in the word are shifted left. Note that these operator names are composed of two minus signs and the `<` or `>` symbol.



Before execution



After execution

Figure 13-15 Operation of −− >

13.5.7 Relational Operators

This group of operators arithmetically compares the top two objects on the operand stack and returns in their place a single Boolean object whose value is either *true* or *false*. This is illustrated in Figure 13-16. The representation of the quantities true and false in the Virtual Machine is arbitrary. Typically the values 1 and 0 are chosen for true and false respectively. For op2 the top object on the stack and op1 the next object, the relational operators are:

EQ	- test op1 equal to op2
NE	- test op1 not equal to op2
LT	- test op1 less than op2
GT	- test op1 greater than op2
LE	- test op1 less than or equal to op2
GE	- test op1 greater than or equal to op2

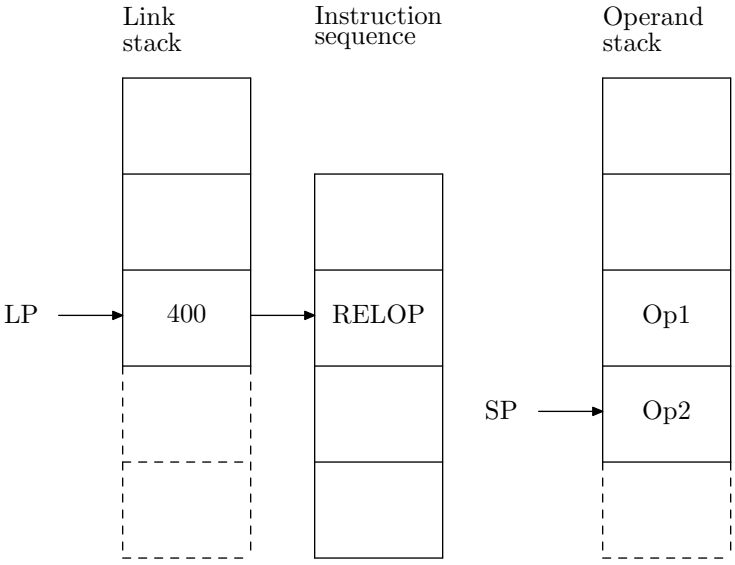
Satisfaction of a relational test returns the quantity true to the stack. Otherwise the quantity false is returned.

13.5.8 Logical Operators

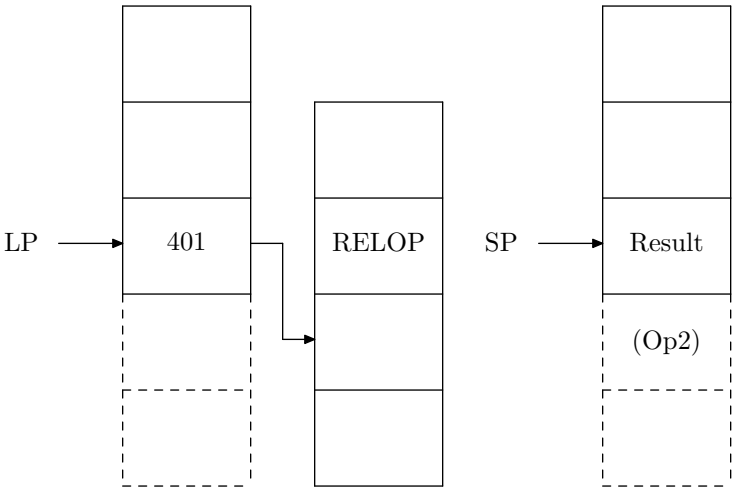
The logical operators are analogous to the binary arithmetic operators, but operate on Boolean operands instead of integer operands. The operators in this group are:

NOT	- equivalent to COMPL
AND	- equivalent to MASK
OR	- equivalent to UNION
XOR	- equivalent to DIFFER

Figure 13-17 illustrates the operation of AND, OR, and XOR. Figure 13-13 illustrates the logic of NOT, substituting a Boolean object on the stack. For all of these operators, the standard rules of Boolean algebra are to be implemented.

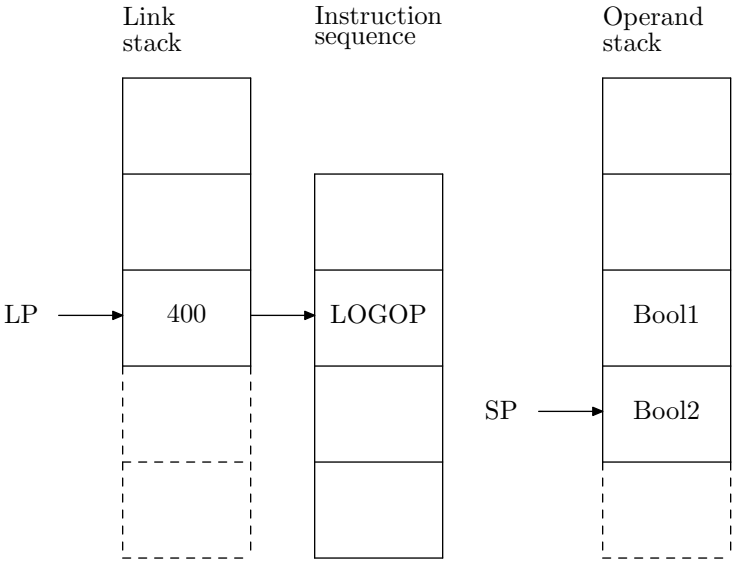


Before execution

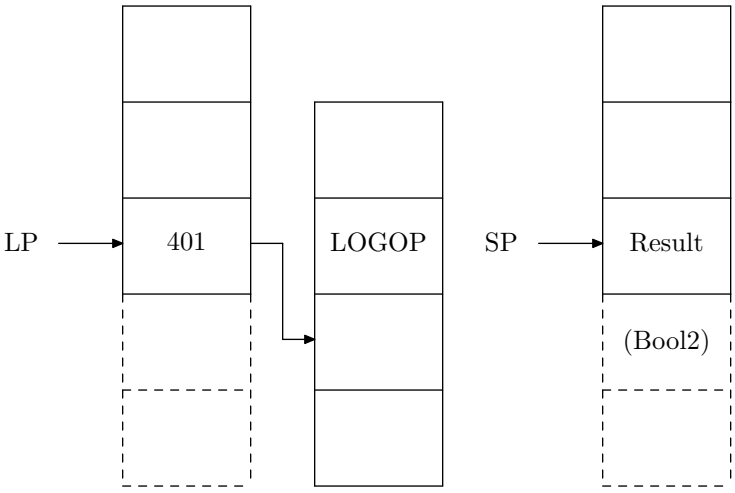


After execution

Figure 13-16 Operation of RELOP



Before execution



After execution

Figure 13-17 Operation of LOGOP

13.5.8.1 The CHOOSE Primitive

The CHOOSE primitive tests a Boolean object on the stack, and selects one of two remaining objects based on the value of the Boolean object. This is illustrated in Figure 13-18. Note the order of the objects on the stack. If the Boolean object is true then op1 is returned as the result. If the Boolean object is false, op2 is returned as the result.

13.5.9 Control Transfer Primitives

This group consists of those primitives that directly affect the value of the top item on the link stack. These are:

GO
YES
NO
TRUE
FALSE

13.5.9.1 The GO Primitive

The GO primitive pops the link stack, pops the top item from the operand stack, and pushes this item onto the link stack. The object on the operand stack is assumed to be a valid virtual address. Figure 13-19 illustrates the operation of GO. Since this operation replaces the top item on the link stack, the address of the GO operator is lost.

13.5.9.2 The YES Primitive

The YES primitive removes two objects from the operand stack. It tests the second object as a Boolean value. If this value is true, the link stack is popped and the first object from the operand stack is pushed onto the link stack. If the value is false, the link stack is not affected and, therefore, execution continues to the next sequential instruction. Figure 13-20 illustrates the operation of YES for a true value case.

13.5.9.3 The NO Primitive

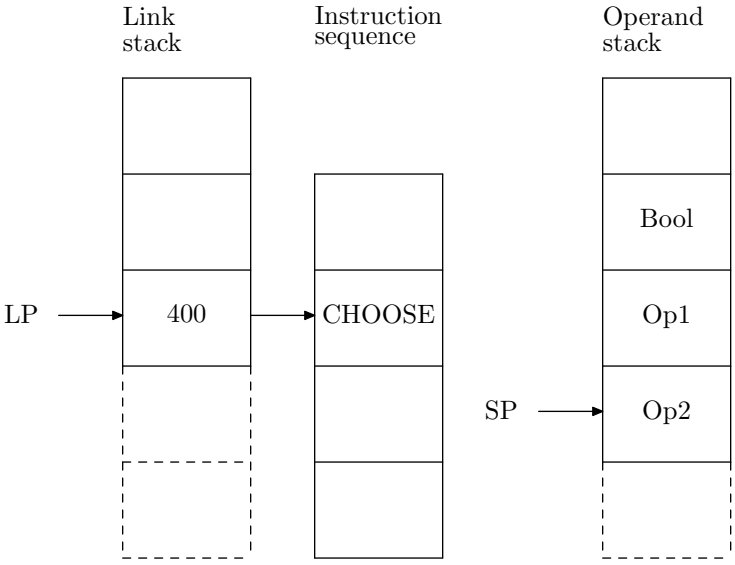
The NO primitive is the converse of YES. It operates on the link stack only if the Boolean object is false. Figure 13-21 illustrates the NO operator for a true case.

13.5.9.4 The TRUE Primitive

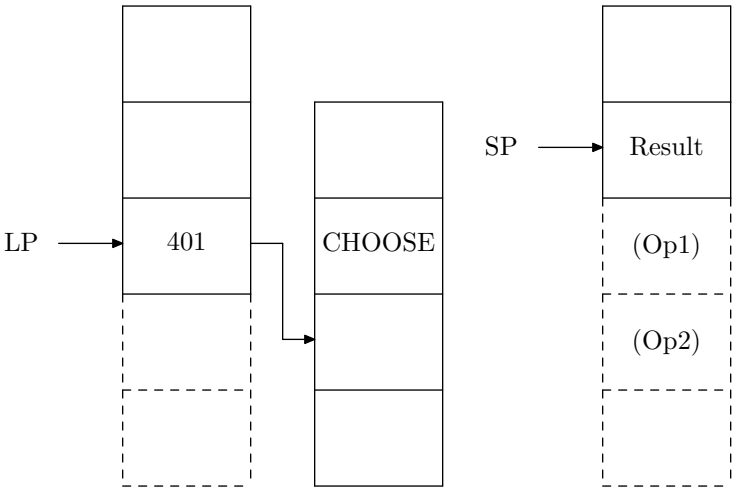
The TRUE primitive provides a conditional skip of a single instruction. It expects a Boolean object on the stack. If this object is true, the next instruction is executed. If the object is false the next instruction is skipped. In any case the object is popped from the stack. Note that since GET and GETV are instructions which are two words long, in these two cases the skip must increment the link stack value by 2. The operation of TRUE is illustrated in Figure 13-22. This Figure is for a non-skip case. See Figure 13-23 for a skip case.

13.5.9.5 The FALSE Primitive

The FALSE primitive operates exactly as the TRUE operator except that the skip occurs if the Boolean value is true, and normal execution continues if the value is false. This is illustrated in Figure 13-23 for a true case. This Figure also shows the special incrementing which is required if TRUE or FALSE are followed by a GET or a GETV.

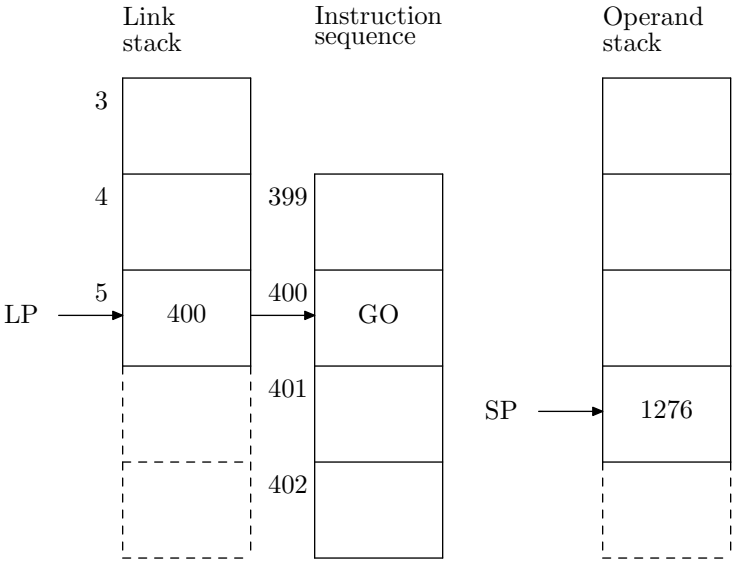


Before execution

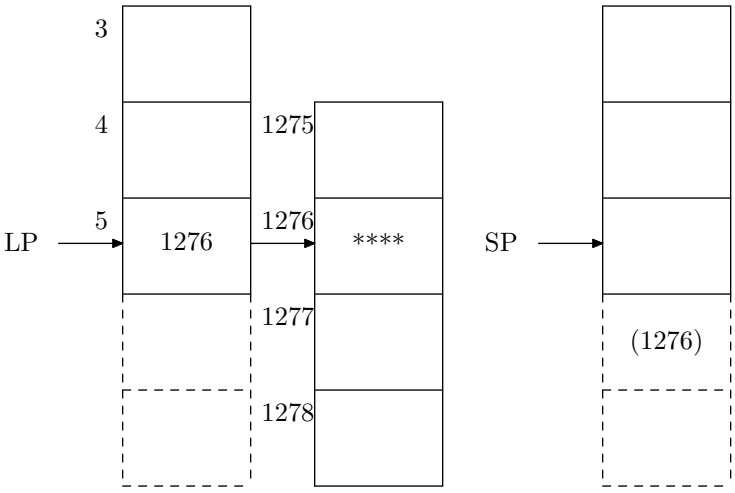


After execution

Figure 13-18 Operation of CHOOSE

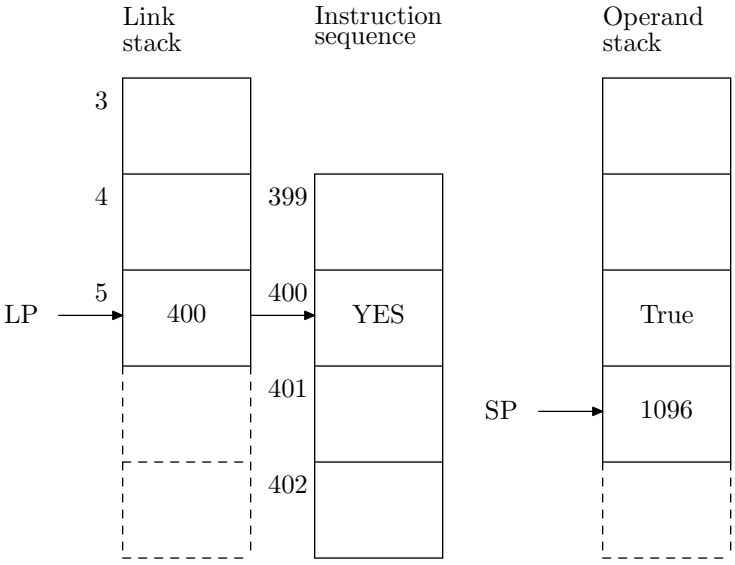


Before execution

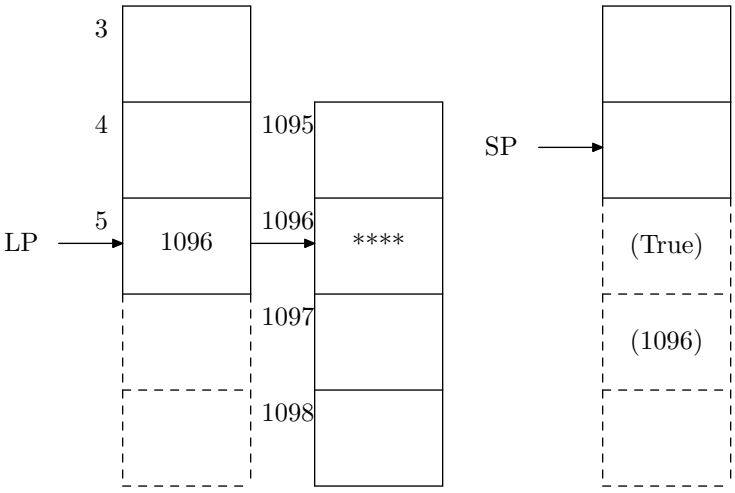


After execution

Figure 13-19 Operation of GO

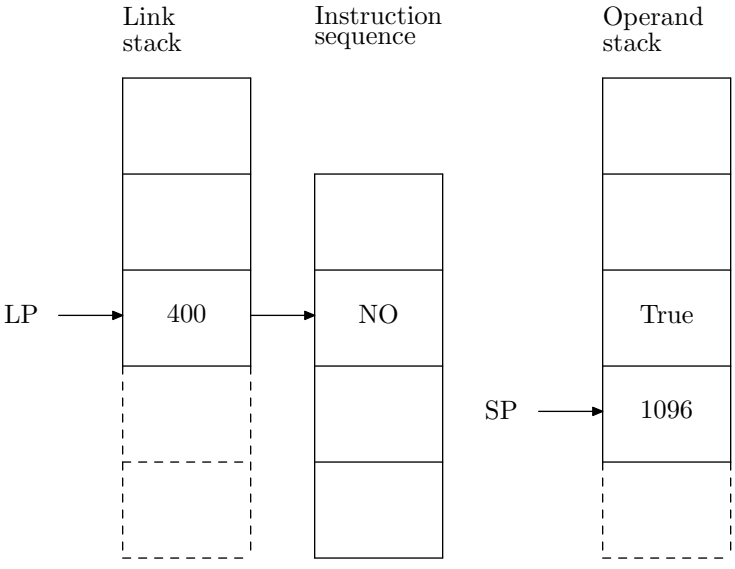


Before execution

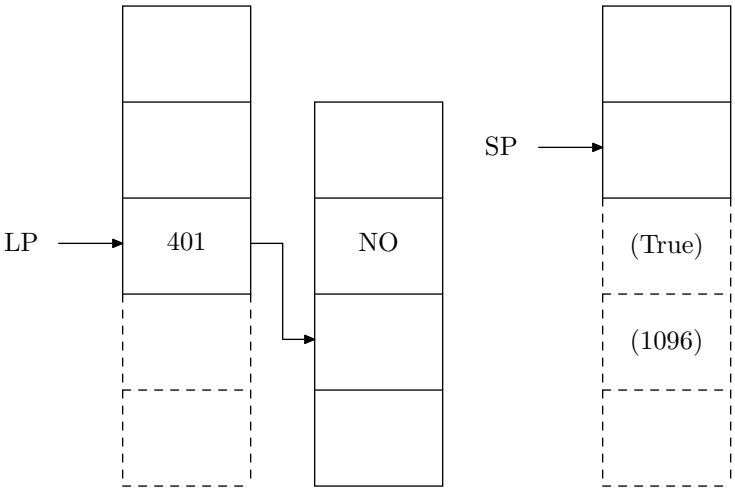


After execution

Figure 13-20 Operation of YES

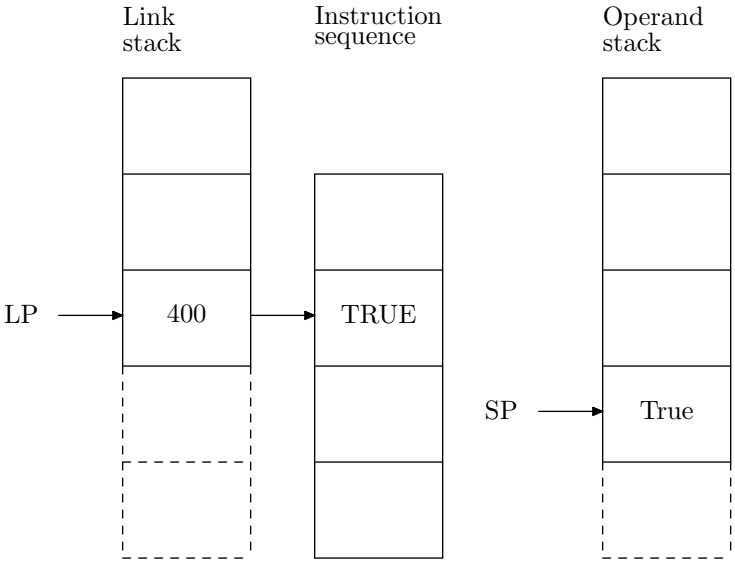


Before execution

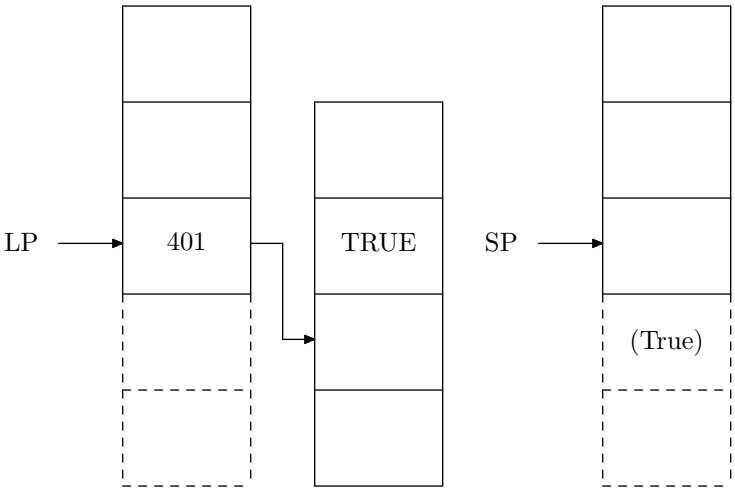


After execution

Figure 13-21 Operation of NO

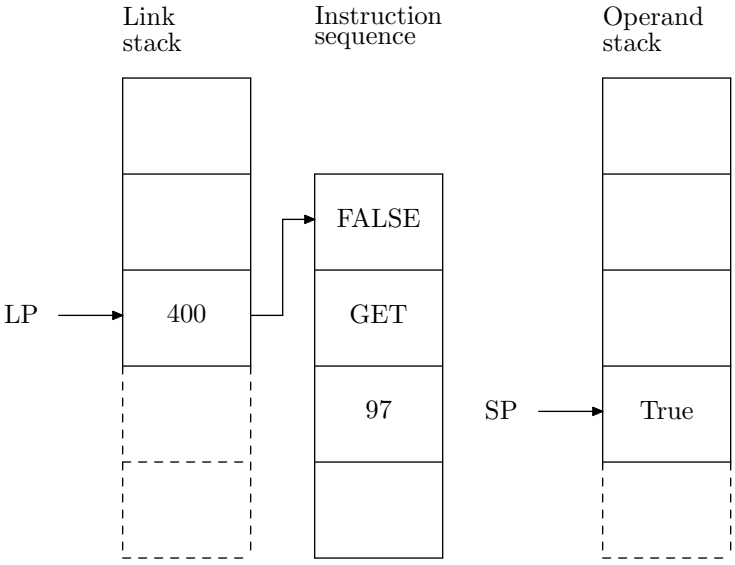


Before execution

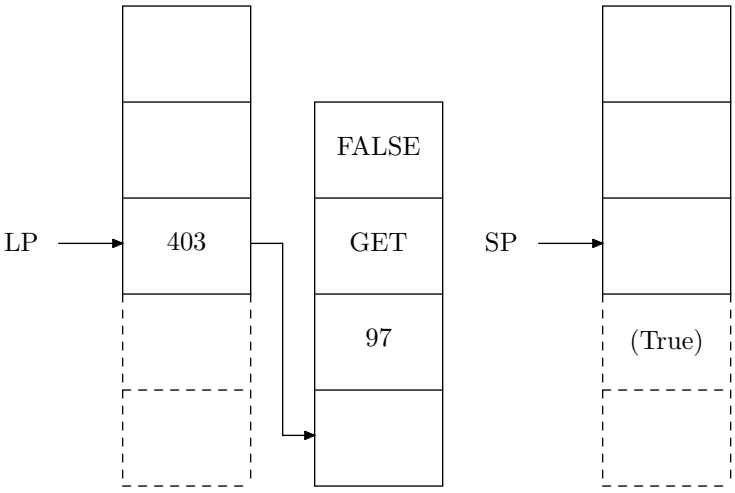


After execution

Figure 13-22 Operation of TRUE



Before execution



After execution

Figure 13-23 Operation of FALSE

13.5.10 Procedure References

A procedure reference is effected in one of two ways:

1. Implicit procedure reference
2. The DO operator.

As mentioned in Section 13.4.4, any op-code with a value greater than 80 is interpreted as a procedure reference. This is termed an *implicit* reference to a procedure. When a procedure is referenced, either implicitly or by means of the DO operator, the procedure address is pushed onto the link stack. Figure 13-24 illustrates an implicit procedure reference. The top item on the link stack is incremented before the new item is pushed onto the stack.

13.5.10.1 The DO Primitive

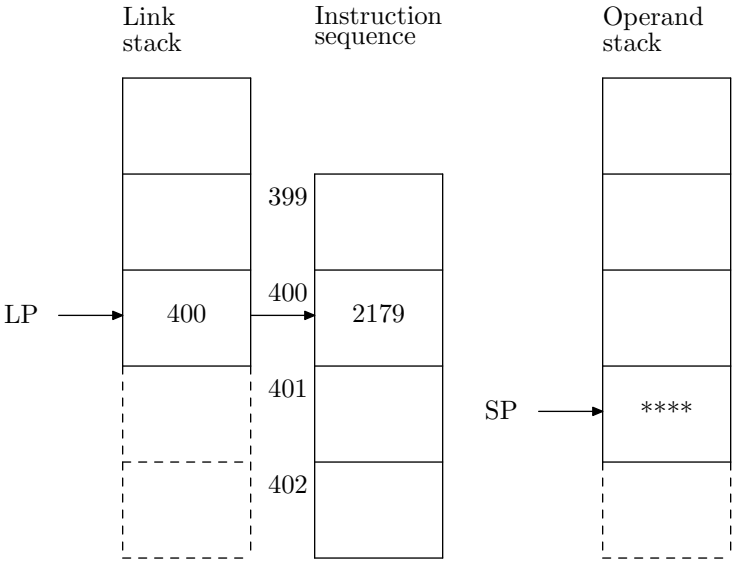
The DO primitive pops the top object from the operand stack and interprets it as a procedure address or, if the value of the object is less than or equal to 80, it interprets it as a reference to a primitive. It then effects an implicit procedure reference using that address, or transfers control to the primitive. Operation for a procedure reference is illustrated in Figure 13-25.

13.5.10.2 The ENTRY Primitive

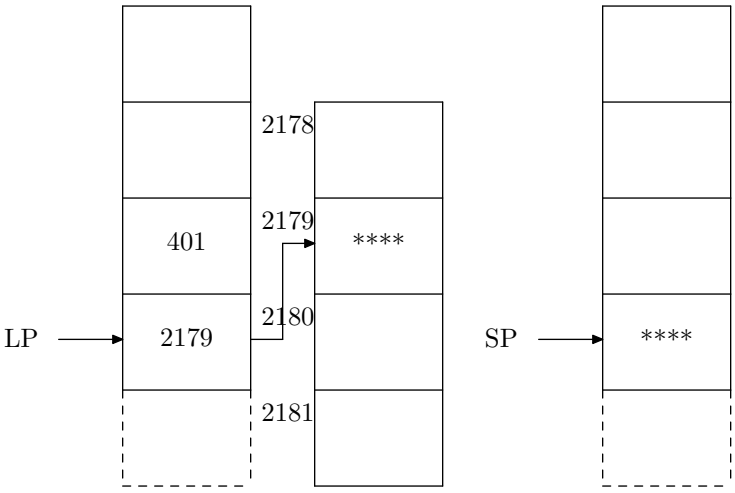
In the current Virtual Machine definition the ENTRY operation does nothing, as procedure references are generated in-line. However, some procedure entry steps, or other computation, may be embodied in the ENTRY operation if the implementer so wishes. A procedure is not correctly written unless its first reference is ENTRY.

13.5.10.3 The EXIT Primitive

The EXIT primitive is used to return from a procedure. It does this by popping the top address from the link stack. This causes the top of the link stack to point to the instruction following the last procedure reference. This is illustrated in Figure 13-26.

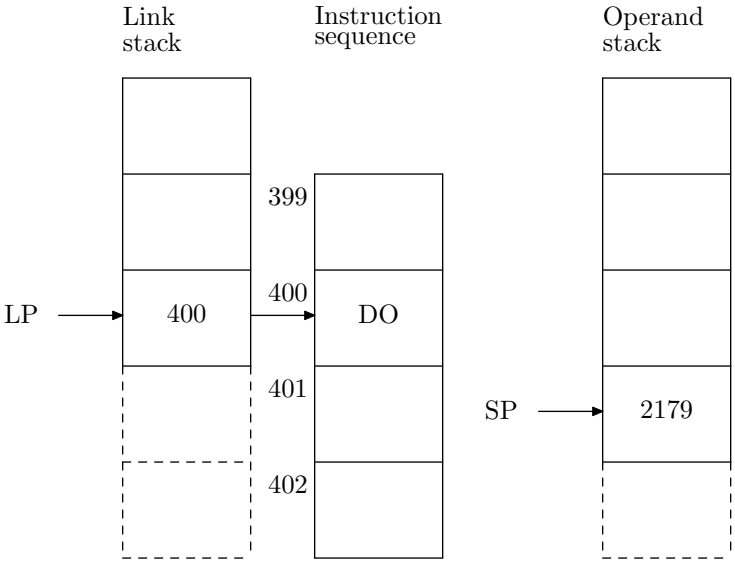


Before execution

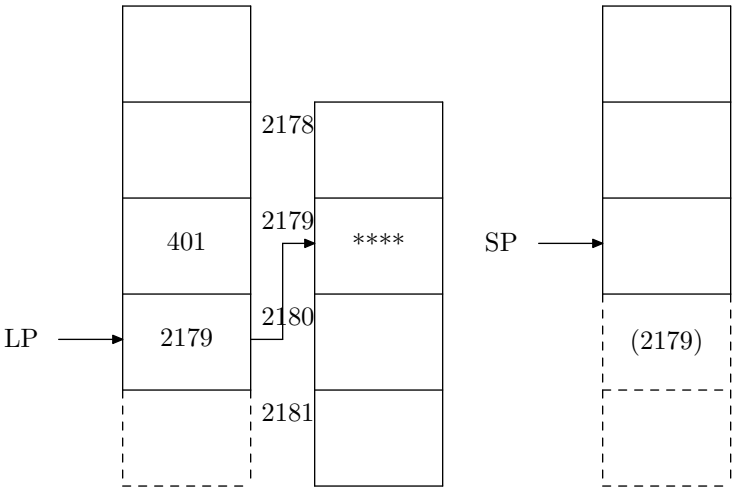


After execution

Figure 13-24 Operation of Implied DO

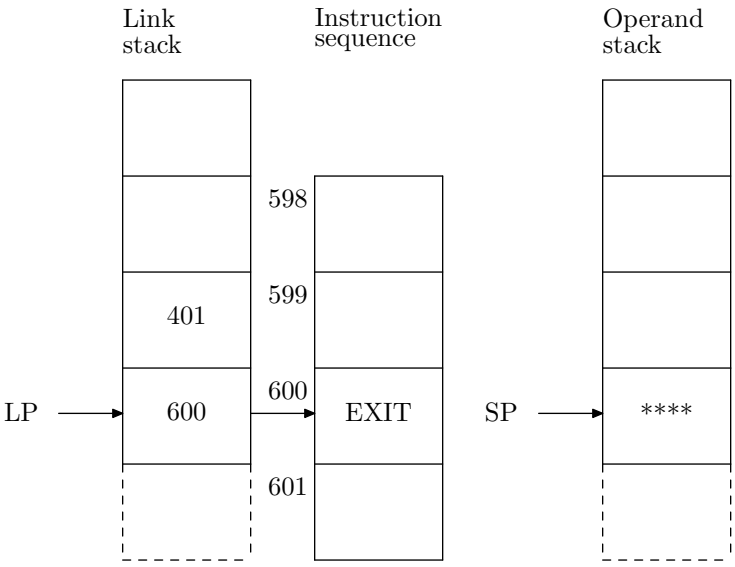


Before execution

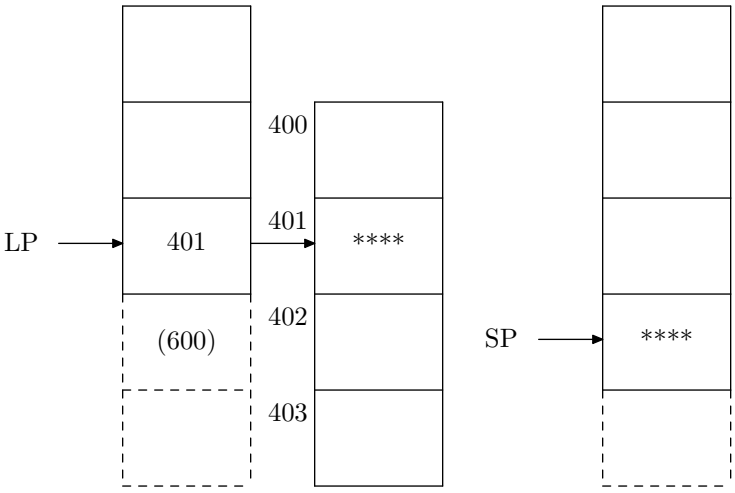


After execution

Figure 13-25 Operation of DO



Before execution



After execution

Figure 13-26 Operation of EXIT

13.5.11 The String Primitives

The string primitives are:

GETCH
PUTCH
MATCH
DICMATCH

13.5.11.1 The GETCH Primitive

The GETCH primitive obtains a character from a MINT string. Its argument is a character address pair (CAP) address. The character to be obtained is pointed to by the index in the CAP. The index is not affected by the operation. If the CAP index value is greater than the length of the string, a zero value is obtained. The operation of GETCH is illustrated in Figure 13-27. In the C implementation, GETCH is provided by a MINT function found in MINTSUBS. It is:

```
GETCH:ENTRY
    DUP->@STRAD, VAL(1 FROM STRAD)->@CHNO, DUP(VAL)->@STRAD,
    VAL() GT CHNO THEN<VAL((1+(CHNO-->2)) FROM STRAD),
        -->((CHNO MASK 3)<--3), MASK 255
    ELSE 0>
    EXIT
```

13.5.11.2 The PUTCH Primitive

The PUTCH primitive stores a character into a MINT string. The two arguments for the operation are the CAP address and the character which is to be stored. The PUTCH operation stores the character using the CAP index to determine the character position within the string. No other characters in the string are affected, and the CAP index is unchanged. The character is always stored, even if the CAP index points beyond the current string length. The operation of PUTCH is illustrated in Figure 13-28. In the C implementation, PUTCH is provided by a MINT function found in MINTSUBS. It is:

```
PUTCH:ENTRY
    <=>, ->@STRAD, VAL(1 FROM STRAD)->@CHNO,
    (1+(CHNO-->2)) FROM VAL(STRAD)->@STRAD,
    VAL(STRAD) MASK VAL((CHNO MASK 3) FROM @CMSK),
    <=>, <--((CHNO MASK 3)<--3), UNION, ->STRAD,
    EXIT
```

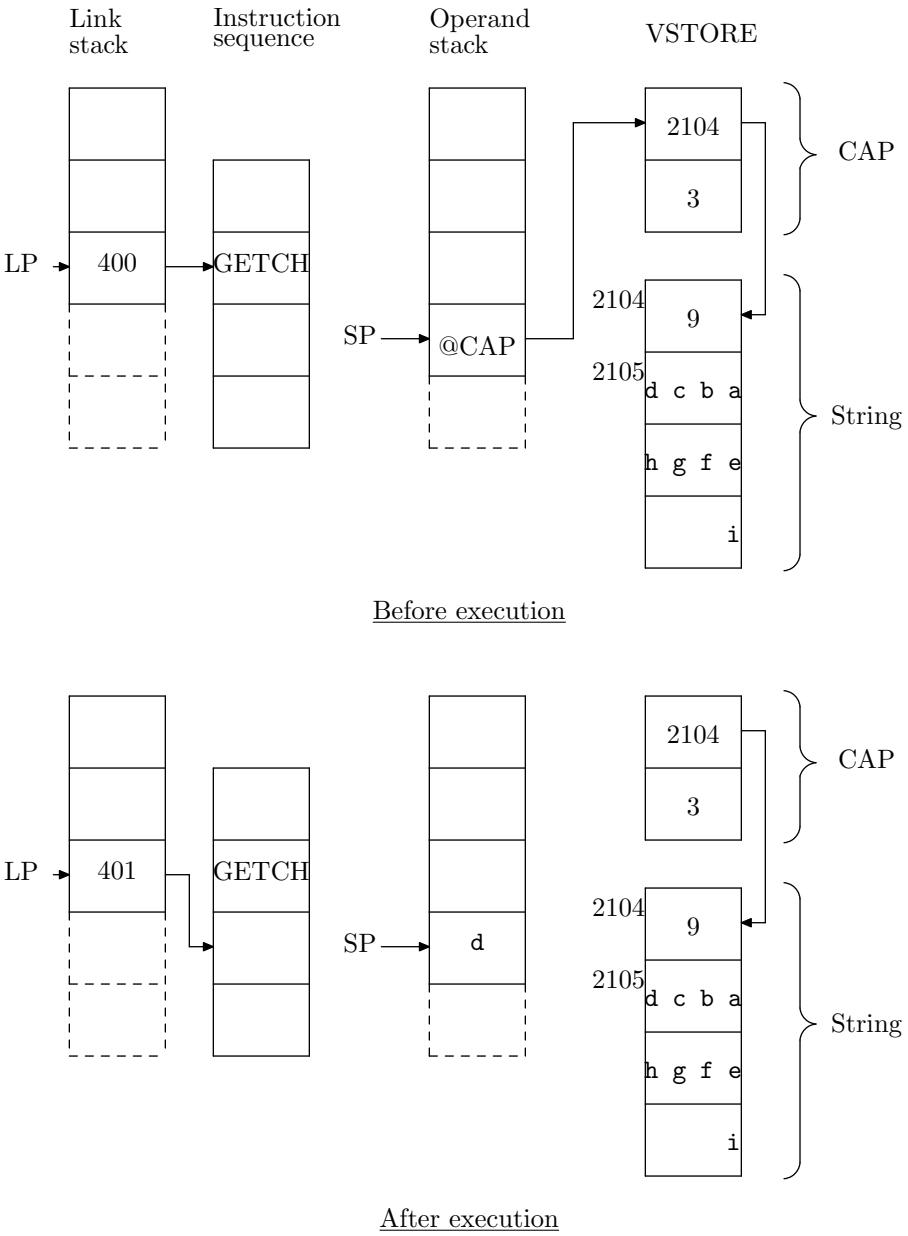


Figure 13-27 Operation of GETCH

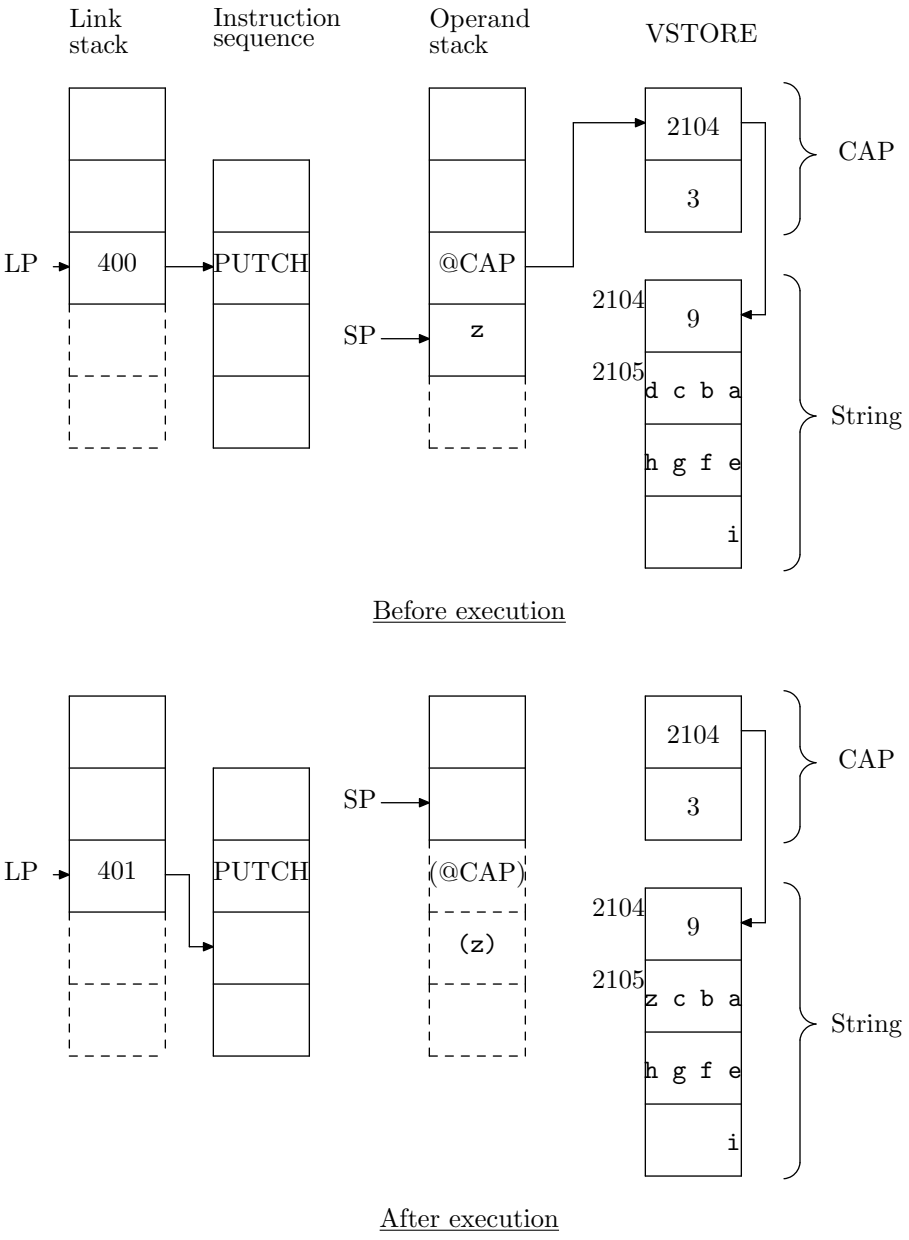


Figure 13-28 Operation of PUTCH

13.5.11.3 The MATCH Primitive

The MATCH primitive matches a key string against a test string addressed by a character address pair (CAP). The CAP and string structures are described in Chapter 10. MATCH has two operands:

1. The address of the key string
2. The address of the CAP.

MATCH returns a single Boolean result which is determined in the following manner. The key string character count from the first word of the string is used as a loop count. In the loop, each character in the key string is compared with each character in the test string. The first character to be tested in the latter is that addressed by the CAP. A match condition occurs when the loop count decrements to zero without a character mismatch. Two conditions can result in a non-match result. First, there may be a mismatch between two of the characters being compared. Second, the end-of-string may be encountered on the test string; i.e. the key string contains more characters than remain in the test string. When a successful match occurs a Boolean true is returned to the stack. In addition, the character index in the CAP is set to the index of the character following the last matched character. In the case of a non-match, a Boolean false is returned to the stack and the character index in the CAP for the test string is unaffected. A match will always occur if the key string length is zero. In this case the character index value in the CAP is unaffected. The operation of MATCH is illustrated in Figure 13-29.

13.5.11.4 The DICMATCH Primitive

The DICMATCH primitive repeatedly applies the MATCH operation to each entry in a list. MINT list format is described in Section 9.2. The DICMATCH operator has three operands:

1. Address of list pointer (listp)
2. Address of list end (listend)
3. Address of test string CAP

DICMATCH operates on the test string in an identical manner to the MATCH primitive. It uses the string in each list record as a key string until a match occurs, or until the end of the list is reached (listend). If

a successful match occurs the DICMATCH operator returns to the stack the address of the previous record to the one that matched; i.e. if record 2 matched the test string, the address of record 1 would be returned; if record 1 matched, the address of the list pointer (listp) would be returned. As with the MATCH primitive the CAP index is updated. In the case of a non-match the address of the last list record is returned to the stack and the CAP index is unaffected. The operation of DICMATCH is illustrated in Figure 13-30.

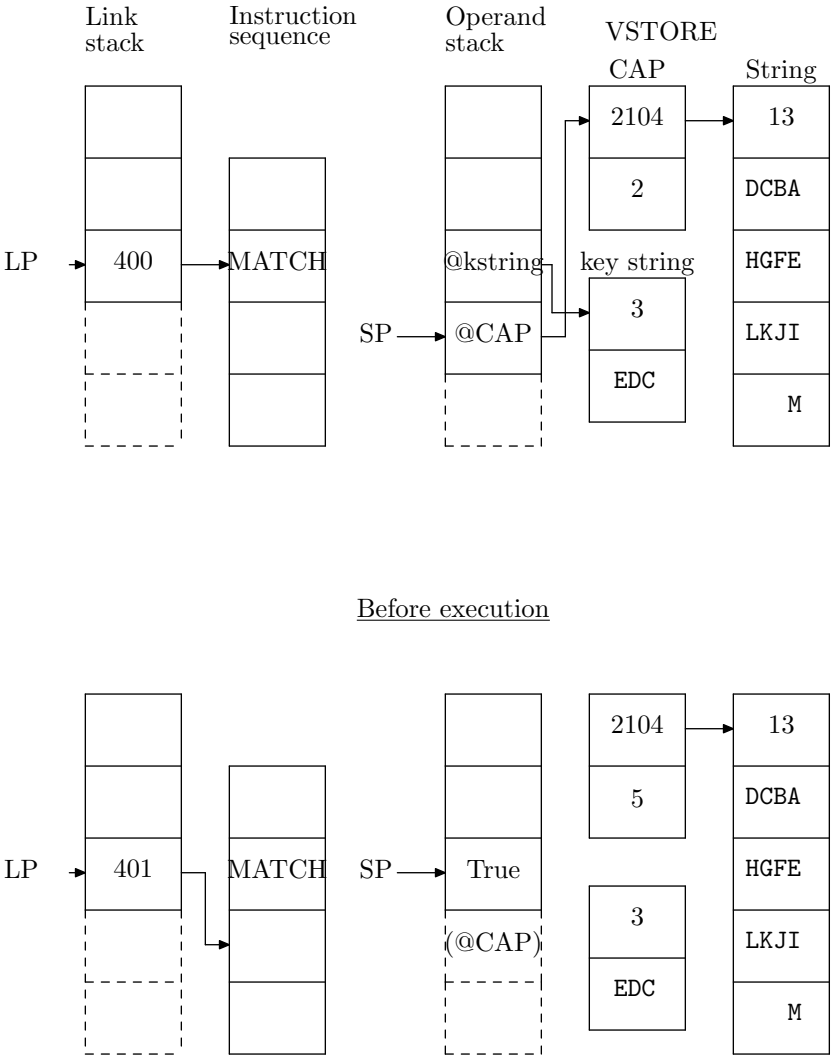
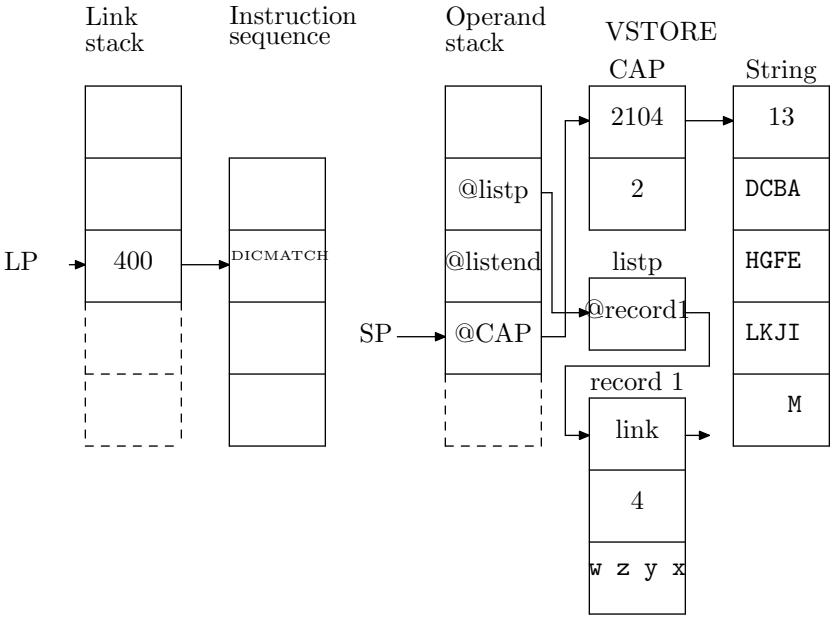
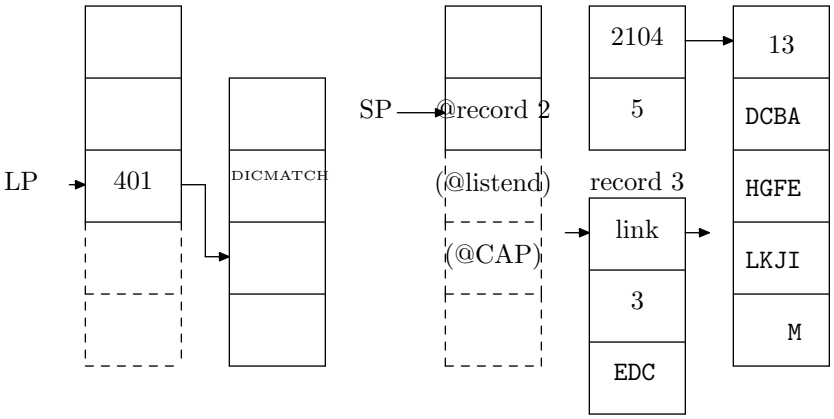


Figure 13-29 Operation of MATCH



Before execution



After execution

Figure 13-30 Operation of DICMATCH

13.5.12 External Interface Primitives

The external interface provides for access to information which is not in VSTORE. In general the containers of such information are referred to as *external segments*. The VM(M) instructions which reference such segments use the segment index which is assigned to each such segment. The main use of this structure is for obtaining information from or transmitting information to files or devices. Both sequential and index-based access mechanisms are provided. The INCH and OPCH operators provide sequential access, while the SEGIO operator provides an indexed record mechanism.

All external segments must be opened before use, except for the primary I/O interface which is always considered to be open for input and output. This primary I/O interface uses segment index zero, and is normally an interactive console device. It is the OPENF primitive which obtains the segment index which is used by all accessing primitives.

13.5.12.1 The OPENF Primitive

The OPENF primitive uses the segment type and the string address supplied on the stack to open a segment for the requested function. It obtains a unique external segment index. The standard defined segment types are given in Table 10-1.

13.5.12.2 The CLOSEF Primitive

The CLOSEF primitive uses the supplied segment index to release the segment associated with that index.

13.5.12.3 The INCH Primitive

The INCH primitive is used to obtain the next character from a specified segment. INCH requires one argument which is the segment index. It returns a single argument which is the next character from the segment. Characters are always produced from a segment in sequential order starting with the first character. If an INCH primitive is referenced after the last character has been produced the Virtual Machine returns the hex value 0x100. This is intended to indicate that no more data are available from the segment, i.e. an end-of-segment condition. After the end-of-segment condition has been reached the 0x100 value should be returned for further INCH references to that segment. A CLOSEF, OPENF sequence should reestablish initial operation. For magnetic tape files, end-of-segment is to

be interpreted as end-of-file. The CLOSEF, OPENF sequence then allows operation on the next tape file. Since the C/R character is used to indicate end-of-string, or end-of-line, the Virtual Machine should generate a C/R character after the last character in a line in cases where the system interface provides line images. Whatever the structure of the system interface, the Virtual Machine should make it appear that lines are separated only by the C/R character. Any additional characters, such as LF or NUL, should only be returned if they are intended to be a part of the input to MINT. The operation of INCH is shown in Figure 13-31.

13.5.12.4 The OPCH Primitive

The OPCH primitive transmits a character to a destination segment. Its operands are the destination segment index and the character which is to be transmitted. Since any character value may be transmitted by the OPCH operator end-of-line may be indicated by transmission of the C/R character. The operation of OPCH is illustrated in Figure 13-32.

13.5.12.5 The SEGIO Primitive

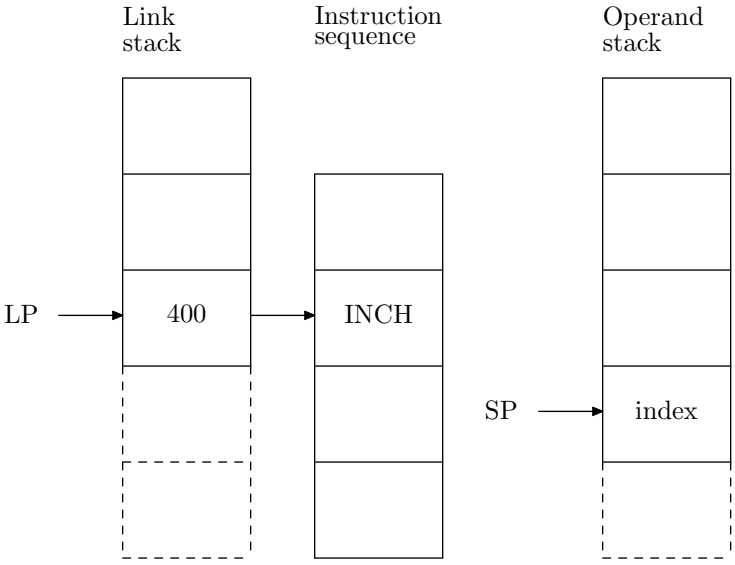
The SEGIO primitive provides a record, or block, structured mechanism for transferring data between external segments and VSTORE. The records are referenced by their index values. Since SEGIO shares the same primitive operation code (39) as OPCH, the VM distinguishes between an OPCH reference and a SEGIO reference by means of the type value supplied when the OPENF operation was performed on the segment. The OPENF type value 3 causes the operation to be treated as a SEGIO reference. The structure of the table whose address is provided on the SEGIO reference is as follows:

Table 13-4 SEGIO Table Definitions

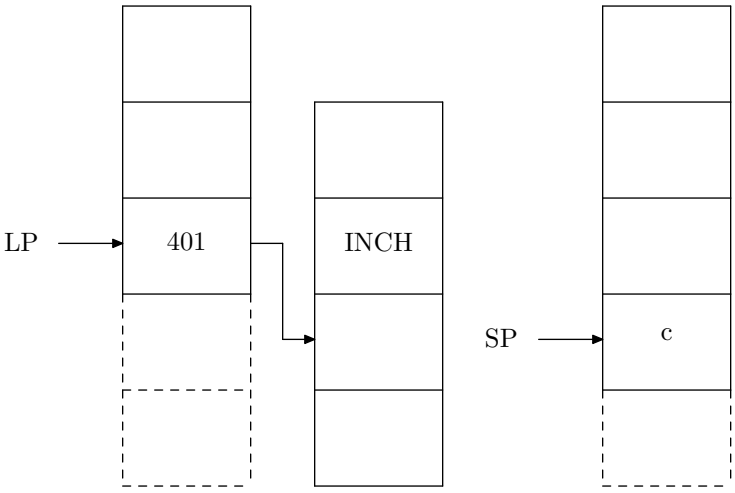
word	meaning
0	source segment number
1	target segment number
2	length of record in words
3	record index
4	VSTORE address of record

The direction of data transfer for a SEGIO operation is determined by the source and target segment index values. If the source index is zero

the transfer is from VSTORE to the target segment. If the target segment index is zero the transfer is from the source segment to VSTORE. Use of SEGIO with both the source and target indices zero or both nonzero is not allowed. The length value in word 2 should remain constant for any specific segment. The record index is defined to have the value 1 for the first record in the segment, and increments by one for each succeeding record.

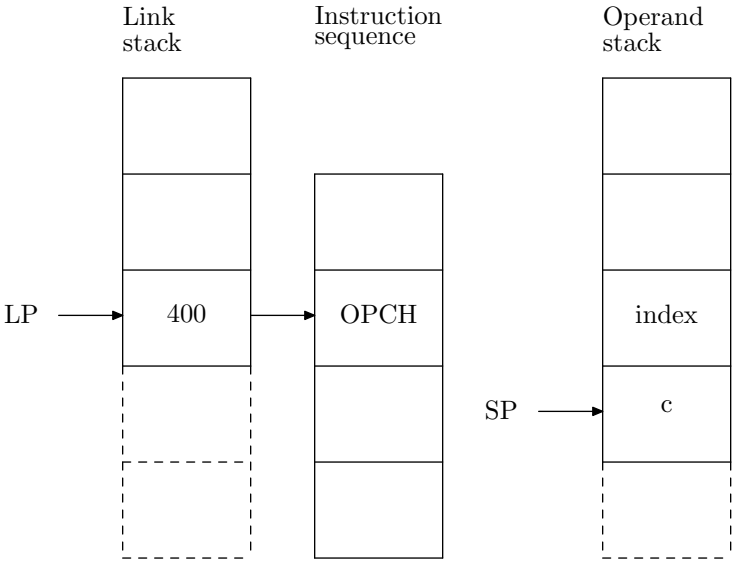


Before execution

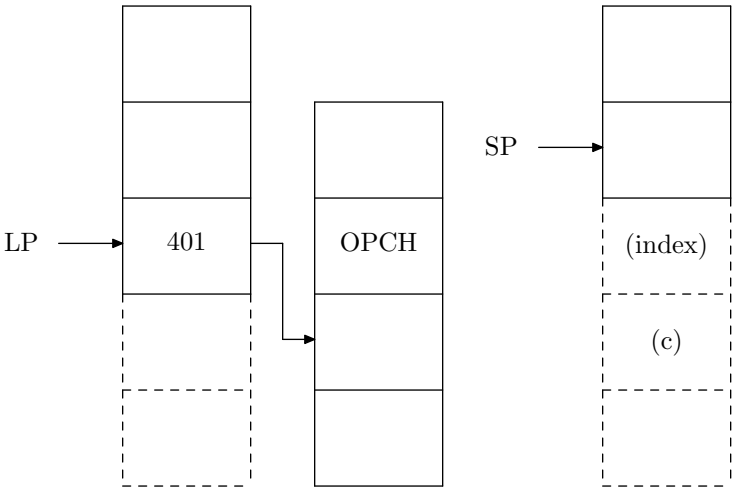


After execution

Figure 13-31 Operation of INCH

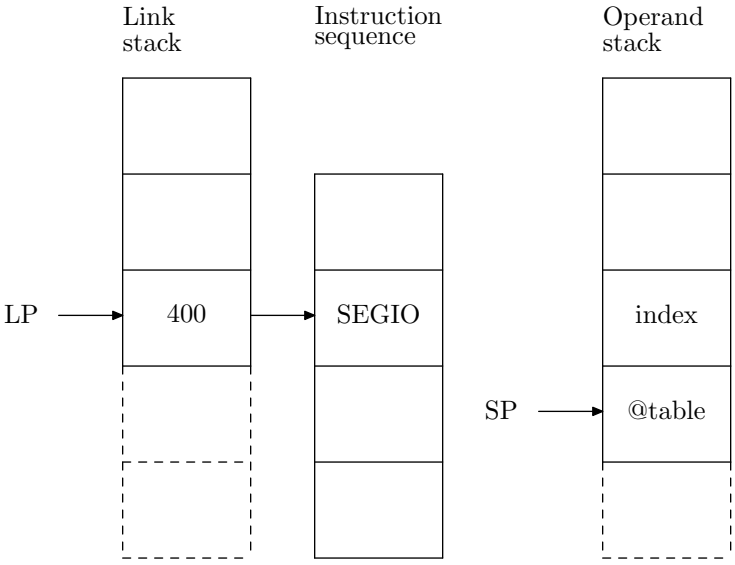


Before execution

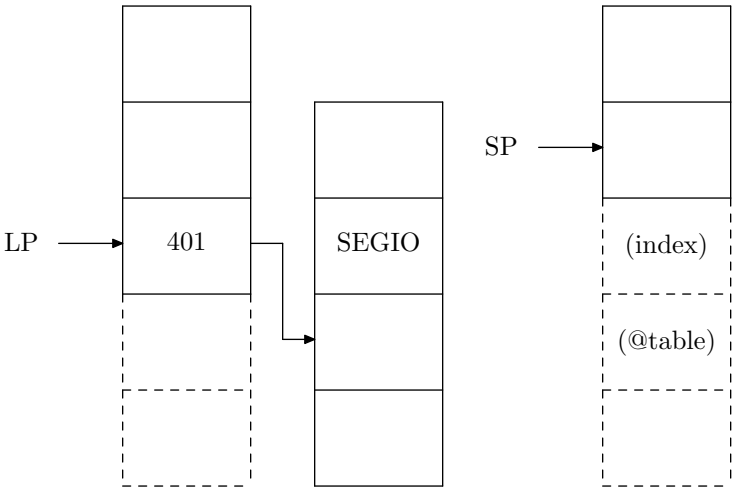


After execution

Figure 13-32 Operation of OPCH



Before execution



After execution

Figure 13-33 Operation of SEGIO

13.5.13 Miscellaneous Primitives

This set of primitives is not mandatory for the Virtual Machine but has been found to be useful. The primitives in this group are:

STOP
ESTOP
PDUMP
TIME
EXR
TRAP
EMULATE

13.5.13.1 The STOP Primitive

The STOP primitive exits from the Virtual Machine. The STOP operator should be used for normal termination conditions. For implementations within an operating system environment this operator should cause a normal return to the system. In stand-alone implementations, STOP should return to a Virtual Machine routine which may then call the VSTORE load routines.

13.5.13.2 The ESTOP Primitive

This operator functions as the STOP operator, but is for error termination conditions. The compiler references ESTOP under error conditions. All implementations should print the current contents of the stacks as a result of an ESTOP reference. It is not assumed that the Virtual Machine can automatically be restarted after a STOP or ESTOP.

13.5.13.3 The PDUMP Primitive

The PDUMP primitive is used to perform a store-image dump of Virtual Store. In many implementations, such a store image will allow faster loading of VSTORE than by means of the portable-format loader. Systems with high speed, large granularity, I/O capability will tend to benefit from the use of PDUMP. It is natural to implement PDUMP as, essentially, a paging structure. PDUMP is not tested by the Virtual Machine diagnostics and is not part of the normal system instruction set.

The CDUMP (See Section 6.6.5) directive uses PDUMP to write VSTORE.

13.5.13.4 The TIME Primitive

The TIME primitive provides the means of obtaining the current time. It is intended that the implementation return the current time in seconds from midnight. However, this primitive is considered optional and may be modified to suit special implementation needs.

13.5.13.5 The EXR Primitive

The purpose of this operator is to allow communication with the host operating environment for purposes which are not provided by the other defined primitives. The implementation of this primitive is optional and is necessarily implementation dependent. Normally a string parameter is supplied. This string is to be interpreted by the host environment.

13.5.13.6 The TRAP Primitive

The purpose of the TRAP primitive is to provide a means of intercepting the execution of VM(M) Virtual Machine instructions. One purpose of such interception is to provide a display of instruction execution. The TRAP primitive is required for the operation of the T$ directive, which provides such a display. The operation of the TRAP primitive is as follows:

Trapping has two state indicators, TRACTIVE (ACTIVE or INACTIVE) and TRON (ON or OFF). If trapping is ACTIVE at the start of the interpreter main instruction processing loop (prior to the instruction fetch and LP register incrementation) the current instruction is not executed. Instead, the interpreter sets TRACTIVE to INACTIVE, pushes the top item on the link stack onto the operand stack, and executes a DO, thus referencing the procedure whose address is on the stack.

There are two actions which may be carried out when the TRAP primitive is referenced. If either trapping is ACTIVE and ON or it is INACTIVE and OFF then the following actions are carried out:

```
DUP -> @TRON

-> @TRACTIVE
```

This, in effect, enables trapping if the argument is non-zero and disables trapping if the argument is a 0. If trapping is ACTIVE and OFF, or if it is INACTIVE and ON then the following actions are carried out:

```
1 -> @TRACTIVE,
restore the top object to the top of the link stack,
unconditionally execute the next instruction.
```

This reactivates trapping after the return from the procedure which was called by the DO operation. Thus, the use of trapping follows these rules:

1. The text which is entered by the DO operator is terminated by a reference to TRAP, with the stack as it was on entry, to cause execution of the intercepted instruction.

2. Trapping is turned on by a reference to TRAP(address).

3. Trapping is turned off by a reference to TRAP(0).

This mechanism minimizes the amount of logic required within the interpreter and allows flexibility of programming of the trapping logic within the MINT system itself.

Uses of TRAP may include instruction usage counts, trapping of instruction or VSTORE references, or any other instruction execution based analysis or control.

13.5.13.7 The EMULATE Primitive

The EMULATE primitive provides the means of replacing current primitives, or implementing new primitives, using MINT text. Any of the operation codes between zero and 79 may be emulated. EMULATE operates on the VM instruction table so that, for the emulated operation code, control comes into the emulation code in the VM instead of going to the specific instruction processing in the VM. The emulation code determines the appropriate action to perform as described below. Two data structures are relevant to the emulation logic. The first is the VM instruction transfer (or jump) table.The second is a table of emulation transfer addresses. Each entry in the instruction transfer table must contain three fields: a field containing the index into the emulation table, a field which indicates if emulation is *active,* and the transfer address field. The emulation transfer table entries contain two fields: the address of the MINT procedure which will execute the emulation and a field containing the normal transfer address copied from the instruction transfer table. This address is saved in the execution table so that it may be restored in the instruction transfer

table when emulation is terminated.

There are two components to the emulation logic. The first is the processing due to execution of the emulation operation code (80). The second is a procedure that is entered due to the fact that its address is set in the instruction transfer table. This second procedure provides the control transfer for the actual emulation.

The EMULATE operator expects two arguments on the operand stack:

- An operation code,
- A procedure address.

The processing due to execution of the emulation operator is divided into three cases:

1. **Procedure address is greater than zero and emulation of this operation code is not currently set**. In this case the following steps are required:
   - Locate a free entry in the emulation table. Store the index of this entry into the index field of the entry in the instruction transfer table for this operation code. This causes emulation action to be *set* for this operation code.
   - Save the address contained in the transfer field of the instruction transfer table entry in the save field of the emulation table entry.
   - Store the address provided on the stack into the emulation table entry transfer address field.
   - Store the address of the emulation control routine into the transfer address field of the instruction transfer table entry.
   - Ensure that the stack pointer reflects the use of two arguments and continue instruction processing.
2. **Procedure address is greater than zero and emulation for this operation is active**. In this case emulation of this operation code is marked inactive and control is returned to the address provided as the parameter.
3. **Procedure address is zero**. This case is used to terminate emulation of the given operation code. The flag and index fields in the instruction transfer table are cleared. The saved transfer address is copied from the emulation table to the instruction transfer table. The emulation table entry is marked as available.

Control comes to the emulation control routine when an operation code is

encountered which has had the address in the instruction transfer table set to point to the control routine.The control routine first tests if emulation is already active for this operation code. If this is the case the original transfer address is obtained from the save field in the emulation table and control is transferred to that address. If emulation is not active the active flag is set, the current value on the link stack is copied to the operand stack (the value is popped from the link stack, and is pushed onto the operand stack), and the address of the emulation procedure (which is stored in the emulation table) is pushed onto the link stack. Then, normal processing is resumed, which will cause the next instruction to be taken from the start of the emulation routine.

This form of control transfer allows emulation to operate without disturbing the link stack or operand stack, and all data are available to the emulation procedure so that it can recover the location of the emulated instruction, can manipulate data on the operand stack, and can reference other procedures.

13.5.13.8 VMDEBUG

The primitive, VMDEBUG, is available to control Virtual machine debug mode. VMDEBUG pops the stack and stores the value in an internal register which determines VM debug mode. If the register is 0 debugging is turned off, and the VM runs at full-speed. If the register is non-zero VM debugging is turned on and, currently, the VM speed is about 1/2 the normal speed. With debugging on the following actions are taken:

1. Address values are checked for the range 80 < address < MAXVSTORE.
2. Instruction values and the current top of stack are pushed into a circular list. In the event of an abnormal condition, the list is displayed and is written to the file: mint-tr.trc.

The option of attempting restart is available after an abnormal stop. On restart, normal entry to the compiler is taken. Of course, this may not always provide useful results.

13.6 Summary of Virtual Machine Primitives

The following Table provides a summary of all Virtual Machine primitives, and their numeric operation codes. The numeric codes are needed to interpret object format MINT code.

Table 13-5. Virtual Machine Primitives

numeric code	instruction	numeric code	instruction
0	*Illegal*	30	YES
1	GET	31	NO
2	GETV	32	GO
3	VAL	33	DO
4	− >	34	ENTRY
5	DUP	35	EXIT
6	LOSE	36	GETCH
7	GLKP	37	PUTCH
8	SLKP	38	INCH
9	+	39	OPCH, SEGIO
10	−	40	MATCH
11	*	41	DICMATCH
12	/	42	STOP
13	NEG	43	PDUMP
14	FROM	44	TIME
15	MASK	45	OPENF
16	UNION	46	*Illegal*
17	DIFFER	47	CLOSEF
18	COMPL	48	EXR
19	EQ	49	TRAP
20	NE	50	<=>
21	LT	51	TRUE
22	GT	52	FALSE
23	LE	53	ADV
24	GE	54	ESTOP
25	NOT	55	ADIFF
26	AND	56	−− >
27	OR	57	< −−
28	XOR	58	VMDEBUG
29	CHOOSE	:	*Illegal*
		80	EMULATE

14. The Distributed MINT System

14.1 Introduction

This Chapter is intended as a guide to the implementation of MINT systems. It discusses the text files that comprise the distributed system. These are:

- Object text file of Virtual Machine diagnostics
- Object text file of MINT auto-compiler
- Source text files of:
  - compiler common routines
  - compiler main text
  - auto-compiler routines
  - Virtual Machine diagnostics
  - trace (T$) routines
  - identifier display (LV$) routine
  - syntax analysis (M-TRAN) routine
  - instruction analysis (TRMIX) routines
  - text editing routines
  - character reversal (REVSTR) routine
  - file management routines
- Listing of compiler
- Listing of diagnostics
- Other documentation

There is also a PDUMP format file, MCOMP.PDM, of the compiler. At this point it is a good idea to run the diagnostics.

It is assumed that a Virtual Machine has been implemented as described in Chapter 13.

14.2 Virtual Machine Diagnostics

The Virtual Machine diagnostic is a stand-alone MINT program designed to exhaustively test a Virtual Machine implementation. The object file is stand-alone in that it can be executed independently of any other procedures. It is quite compact, as only the test routines are included. None of the normal compiler facilities are needed or available. The object-text format is as described in Section 13.3. It should be loaded and initiated by:

mint -f MINTDGPF

An error-free execution will produce output as shown below:

```
0
1
2
3
4
5
6
7
8
VM(M) diagnostics pass one completed.
GET
GETV
VAL
STORE
DUP
LOSE
EXCHANGE
ADV
ADD
SUB
MUL
DIV
 - Remainder
NEG
FROM
ADIFF
MASK
UNION
DIFFER
COMPL
NOT
EQ
NE
LT
GT
LE
```

```
GE
AND
XOR
RIGHT-SHIFT
LEFT-SHIFT
OR
CHOOSE
YES
NO
TRUE
FALSE
GO
DO
 - nested references
GLKP,SLKP
MATCH
 - character
 - pointer
 - Odd, even, offset strings
DICMATCH
 - string
 - single character
 - pointer
Type a line of text to be echoed by INCH.
This is a test line for INCH.

Check for correct echo.
VM(M) diagnostics completed.
ESTOP instruction executed. Diagnostics follow.
```

The diagnostics operate in two phases. The first phase briefly validates the operation of each instruction required to perform the phase two tests. As each stage of phase one completes successfully a single digit is printed. Failure to print a digit indicates failure of that step. The tests in each step are as follows:

0	initial test of the operation of GET, GO, OPCH and function referencing
1	tests the operation of GO
2	tests the operation of NE and NO on a supposed *false*
3	tests the operation of NE and NO on a supposed *true*
4	tests the operation of GETV
5	tests the operation of `->`
6	tests the operation of VAL
7	tests function referencing
8	tests the operation of GETCH

If the digit 0 fails to be printed on loading the diagnostics, the instructions GET, GO, and OPCH should be investigated; if these appear to be

correct, the portable-format load routine or the operation of the Virtual Machine main instruction loop should be suspected. The second phase of the diagnostics exhaustively tests all the instructions in the Virtual Machine. Before each instruction is tested its name is printed. Any subsequent diagnostics refer to the last printed instruction. Each instruction is tested many times with different operands. For each execution the obtained result is tested. If it is incorrect the diagnostic:

```
**** failed ****
```

is printed. In addition, the status of the operand stack is tested to verify that the correct number of objects have been removed or returned. If an error occurs in the operand stack logic the diagnostic:

```
**** stack exception ****
```

is printed. The diagnostic routine attempts to continue processing after an error so that multiple errors may be detected. However, errors may propagate since the diagnostics tend to assume that previously tested instructions are operating correctly.

14.3 The Compiler Object File

Once the Virtual Machine diagnostics have been executed successfully the compiler object file itself may be loaded using:

```
mint -f MINTPFA
```

The compiler should sign-on with the message:

```
MINT-3 Virtual Machine (32-bit Virtual Memory): Version 1.2.3
VSTORE size 16384K words. Start PF load.....:
MINT-3 System: Version 3.0. Created on: 060714
Copyright D.F. Hendry, 1990, 2004
VM>
```

At this point the compiler is ready for source input.

It may be necessary to adjust the size of the system to fit a particular implementation. Two parameters govern the amount of the compiler's working area. These are the maximum address of data-space (MAXDS$) and the maximum address of procedure-space (MAXPS$). These two values are computed and set by the AUTO directive (See Section 14.6.1.1).

They are used by the compiler to initialize VSTORE pointers. Since these variables occupy locations 1 and 2 of VSTORE they occur at the beginning of portable-format object-text. The current values of these variables in distributed compiler and auto-compiler systems is 256 for MAXDS$ and 256 for MAXPS$. These values may need to be changed to fit particular configuration requirements. It is not simple to change the portable-format text due to the need to recompute the image checksum value. The simplest approach is to set the required values at the end of the VSTORE load process. Table 1-1 shows the amounts of VSTORE required for the standard system. This may be used as a guide in determining the appropriate values for a particular system. Note that if it is intended to be able to recompile the compiler and auto-compiler somewhat more than twice the space shown in Table 1-1 will be required as the auto-compiler and the new copy of the compiler must fit in VSTORE at once.

14.4 Additional Source Text Files

The system is distributed with a set of source files which may be optionally included into the system as needed. These files are described elsewhere and are itemized at the beginning of this Chapter.

14.5 Character Order Reversal in Strings

The directive REVSTR is available for carrying out the character order reversal which may be required if portable-format text is to be moved between implementations which differ in character order within VM words.

14.6 Compiler Creation and Source Structure

The MINT compiler is written in MINT and can be generated by the compiler. The compiler must be in AUTO mode to compile itself. AUTO mode is set by a reference to the AUTO directive. A compiler which includes the AUTO directive, and supporting routines, is referred to as an auto-compiler. The auto-compiler creates an output file which is an object version of the new compiler. This new compiler can be loaded under any MINT system. When the compiler is in AUTO mode additional facilities are available to provide relocation of address values and to provide object-text output. A single set of source modules is used to create all versions of the compiler.

14.6.1 Auto-compiler Directives

14.6.1.1 The AUTO Directive

The AUTO directive creates and initializes the necessary tables and object text so that all subsequently compiled text becomes a part of the system to be created. As part of this initialization, object text is executed which prevents any subsequently introduced directives or macros from being called upon. Thus, if any directives are to be obeyed during the compilation they must be introduced prior to the execution of the AUTO directive. The AUTO directive also insures that the label SYSSTART has been introduced, but not yet set. At the completion of the compilation this label is set to the start address of the new system. The AUTO directive obtains SIUNIT and then resets SIUNIT to zero. It then reads three images. The first image should contain the intended value of N_d and the second the intended value of N_p. (See Figure 1-2 and Section 14.3 concerning these values.) The third image should contain the identification of the output as a string. After this input has been read SIUNIT is reset to its value which was obtained on entry to AUTO.

The AUTO directive may be used to create any new, self-contained, program. For example, AUTO is used to create the VM diagnostics in portable-format object text form. This is accomplished using the sequence:

```
        MACRO ASUBS:'SI MINTAUTO'
        MACRO PFOUT:'SI MINTPFOT'
UNLOCK INTDIC
DICT DIAGD:HDICT
        ASUBS
        PO MINTDGPF.NEW
        AUTO
32768
3000
Diag
        SI MINTDIAG
        GENPF,
```

This text is included in the source files as CDIAG.

14.6.1.2 The PO Directive

The PO directive is defined in the auto-compiler in a manner similar to the standard SO directive. The supplied IPAR-expression for the PO directive is the file name to which the generated object text is written in portable

format. Only text generated after the AUTO directive is written. If a new compiler is generated (using GENSYS) the label PROGEND provides the end address for the text output.

14.6.2 Compiler Source Structure

The compiler and auto-compiler source text have been organized into several modules or elements, each of which is briefly described below.

14.6.2.1 The MINTSYS module

The MINTSYS module is the main source module for the creation of the compiler system. It carries out the following functional steps:

1. It inserts the module MINTCNF which defines the specific modules to be used in building the system.
2. It unlocks INTDIC.
3. It references ASUBS which inserts MINTAUTO.
4. It uses PO to set the output file name.
5. It references the AUTO directive.
6. It sets the system start addresses, and version.
7. It references the macros IOSBS, SUBS, COMP, and ODIR, which insert the compiler source files.
8. It declares PROGEND to set the end of the complied code.
9. Finally, it references GENSYS to create the new system.

The following shows the MINTSYS module as used in the current (3.0) implementation:

```
   SI  MINTCNF
PAGE
   UNLOCK INTDIC
   ASUBS
   PO  MINTPFA.NEW
   AUTO
32768
32768
3.0
   PROG
FN SYSSTART , SYSSTART,
   IOSBS
PAGE
```

```
   SUBS
   COMP
   ODIR
PROGEND: , FORGET PROGEND,
   GENSYS
```

14.6.2.2 The MINTCNF module

The MINTCNF module defines the locations of other compiler source modules by defining the macros IOSBS, SUBS, COMP, PRIM, ODIR, and ASUBS. These macros appear in MINTSYS.

The following shows the MINTCNF module:

```
.      Create MINT compiler system
.
.      Standard MINT elements
.
MACRO IOSBS: ' '
MACRO SUBS:  'SI MINTSUBS '
MACRO COMP:  'SI MINTCOMP '
MACRO ASUBS: 'SI MINTAUTO '
MACRO PFOUT: 'SI MINTPFOT '
.
.      System dependent elements
.
MACRO PRIM:  'SI MINTOPRM '
MACRO ODIR:  'SI MINTODIR ' .
```

14.6.2.3 The IOSBS Macro

The IOSBS macro is used to introduce functions referenced by subsequent modules which normally are expected to be primitives. Thus, for example, if OPENF is provided as a MINT function, it can be introduced in IOSBS and then referenced subsequently in the compilation process.

14.6.2.4 The SUBS Macro

The macro SUBS inserts the file MINTSUBS, using the SI directive (See Section 10.5.1). MINTSUBS contains all the general purpose functions which are of use in generating any system.

14.6.2.5 The COMP Macro

The macro COMP inserts the file MINTCOMP. MINTCOMP contains the text for the main body of the compiler. COMP also references the PRIM macro.

14.6.2.6 The PRIM Macro

The macro PRIM inserts the file MINTPRIM which contains all optional primitives. The basic set of primitives are defined in MINTCOMP.

14.6.2.7 The ODIR Macro

Directives such as T$ or LV$ which are optional, or other directives which may be implementation dependent, may be defined in the file inserted by ODIR. In addition, functions introduced in IOSBS can be defined in ODIR.

14.6.2.8 The ASUBS Macro

The ASUBS macro inserts the file MINTAUTO which contains the procedures required for an auto-compilation. MINTAUTO contains a reference to PFOUT.

14.6.2.9 The PFOUT Macro

The PFOUT macro inserts the file PFOUTPUT which contains the text required to write contents of VSTORE to an external segment in portable format.

14.7 System Generation Sequences

Using the C implementation as an example, the following sequence will create a new compiler in portable format in the file MINTPFA.NEW:

```
mint
SO  MINTALOG.NEW
TITLE(66,1,'Compilation of MINT Compiler     '), LOCS
SI  MINTSYS
OBREAK
STOP!
```

This new compiler can be written in PDUMP format by:

```
mint  -f MINTPFA.NEW
CDUMP 'MCOMP.PDM' 32768
STOP!
```

If this new MCOMP.PDM is in the MINT home directory the next *mint* command will load it. Or, it may be loaded by:

```
mint -f MCOMP.PDM
```

15. The C Implementation

15.1 Introduction

This implementation is the "reference" implementation of the MINT Virtual Machine. It is written in portable C and should compile and run on any ANSI-C compliant system. Minor changes may be needed to the makefile used for compilation to define the name of the compiler, for example. Makefiles for Linux (makefile.gcc), OSX (makefile.osx), and a few other systems are provided. The system has been tested on current Linux platforms, OSX, and OS/2. For other systems, a few changes to definitions in mdefs.h or simple changes in the makefile may be required. For most applications the C virtual machine should provide adequate efficiency and facilities. In some specialized applications, such as embedded systems, an assembly language coded virtual machine, which may or may not require an operating system, may be appropriate.

Only the Virtual Machine is written in C. Everything else is written in MINT and becomes operational as soon as the Virtual Machine has been compiled correctly. The MINT-coded diagnostics provide a thorough test of the Virtual Machine to ensure that it is correct. If the C source is used without changes, the only reason for the diagnostics to fail is a problem with the compiler used, options given to the compiler, or an error in the runtime libraries. The source language for the system is arranged in nine files. These are:

Name	**Content**
mmain.c	Main program
mvm.c	VM instruction interpreter and all VM instructions except the I/O instructions
mio.c	VM I/O instructions and I/O interface code
mload.c	Portable and PDUMP format loader
mutil.c	Miscellaneous procedures
mdefs.h	Configuration parameters and definitions
proto.h	Procedure prototypes
mtext.h	Global declarations
mvars.h	Data structures

15.2 VM Components

The resources and processing requirements for the VM are available in C and its usual supporting Operating Systems, such as Linux. The VM makes use of stream input/output procedures for I/O, and *malloc( )* to acquire the VM virtual memory space. This simple design makes it easy to port the VM to practically any system.

15.2.1 Storage

The MINT VSTORE is allocated by means of a single *malloc( )* system call. The amount of memory requested is configurable at compile-time. Nearly all systems now provide *malloc( )* in the form that the memory requested is not physically allocated until used. For this reason the default initial memory request is large. If this causes a problem, the default can be reduced.

15.2.2 Stacks

The stacks are implemented as (the current default size is 100 words) blocks of storage. Stack space for both stacks is obtained using *malloc( )*. The value returned by *malloc( )* is used as the stack pointer for each stack. The current default size is 100 words for each stack. The current implementation checks for stack underflow or overflow. If the operand stack overflows, additional space is obtained by use of *realloc( )*. Link stack overflow or a NULL return from *realloc( )* causes the VM to exit.

15.2.3 I/O Management

The two main tasks of the I/O Management section are to provide string format conversion, and to manage the file I/O interface.

15.2.4 The Basic Interpreter

The basic interpreter has two main components, the code for instruction fetch and decode, and the code for each instruction. When the Virtual Machine is started, memory is allocated, MINT code is read into VSTORE, and control is transferred to the beginning of the instruction fetch and decode loop. This code first tests for special instruction trap mode handling. If this is required, the trap code is entered. Otherwise, the value of the instruction which is pointed to by the current top of link stack register is

obtained and executed. If the current instruction value is greater than 80, it is treated as a procedure address and an implied DO is performed. If the value is less than or equal to 80 a case statement executes the correct VM(M) instruction. When each instruction is completed, a jump is taken to the start of the instruction loop. This processing continues until either a STOP is encountered, or until an end-of-file condition is found on stream *stdin.*

Due to the ambiguous nature of C syntax and semantics, there is a compile-time parameter which controls compilation of the virtual machine instruction processing code. This parameter (NDPC) selects the use of expressions which are efficient but which depend on "reasonable" ordering of subexpression evaluation. Normally, this parameter is not defined, and separate expressions are used to force correct order of evaluation. If NDPC is defined it is important to run the VM diagnostics to determine if your compiler has compiled working code.

15.2.5 VSTORE Addressing

VSTORE addressing is quite simple as there are very few points at which the VM(M) Virtual Machine actually makes VSTORE references. There are five storage access primitives: GET, GETV, VAL, `->`, and ADV. In addition, each of the six string primitives makes references to VSTORE. The loader stores into VSTORE when either portable or PDUMP format data are loaded. Finally, the main instruction loop references VSTORE in order to obtain each instruction.

15.2.5.1 The Portable Format Loader

The Portable Format Loader carries out two functions. It loads VSTORE from a portable format source file, and it then sets required values in low VSTORE. Portable format, and the portable format load process are described in Sections 13.3 and 13.4. The values which must be set in low VSTORE are given in Table 13-1. The portable format loader expects its input to be in the file which was named on the command line.

15.2.5.2 The PDUMP Loader

The PDUMP load routine uses a header containing the operand and link stack sizes and contents as written by the CDUMP routine. A version field is written at the beginning of the PDUMP file in order to permit updates to the file format while retaining the ability to read old files.

15.2.6 Primary I/O Interface

The primary I/O (stdin, stdout) interface is quite simple. When an INCH instruction is executed referencing segment zero and no input has been read a *readline( )* call is used to obtain the next line of text and the first character is returned. On each subsequent INCH the next character is returned until the end of line. Then a new line is read by *readline( )*.

Use of *readline( )* permits editing of the input line and provides the other readline features such as input history. The history file is saved in the user home directory as *.mint_hist* and is read in when mint is executed so that previous line history is available.

In case the *readline* library is not available, the VM can be configured to use *getchar( )*.

OPCH with a segment index zero causes characters to be written to *stdout* using *fputc( )*.

15.2.7 File I/O Interface

The file I/O interface provides access to files in the operating system's native file system. The description below is for Linux-like systems. It is a simple matter to provide similar facilities using other operating systems.

15.2.7.1 Sequential Input File Open

A sequential input file is opened by a reference to *fopen( )* with an mode of "r" and a string argument which is the filename passed by the MINT code. If this *fopen( )* fails, an *fopen( )* using "w+" is attempted. If that fails a new filename is requested from the user.

15.2.7.2 Sequential Output File Open

Sequential output files are opened by a reference to *fopen( )* with mode "w+" and string argument which names the file to be opened. Note that if the file previously existed, this operation deletes any contents.

15.2.7.3 File Reading and Writing

The file interface for the INCH and OPCH functions uses the same string conversion routines as used for the primary I/O interface. For reading (INCH), *fgetc( )* is used to obtain the next sequential character. Compo-

sition of the characters into MINT strings takes place within MINT. For writing (OPCH), *fputc( )* is used.

15.2.8 Diagnostic Services

There are three main areas of diagnostic checking within the Virtual Machine implementation. These are stack checking, VSTORE address bounds checking, and the illegal VM(M) instruction trap. In addition, System signaled errors are processed by the Virtual Machine so that relevant Virtual Machine state information may be printed before Virtual Machine termination. There is a standard error termination display of the Virtual Machine state. This display prints the locations in VSTORE of the last 20 executed instructions, contents of the current operand stack, and the contents of the current link stack. This display is normally preceded by a specific diagnostic message which depends on the error condition.

15.2.8.1 Stack Checking

The standard stack checking is link stack underflow and overflow check. This check prevents stack errors from corrupting other data, and it is relatively inexpensive to check.

However, the system may easily be configured with this checking disabled.

15.2.8.2 VSTORE Bounds Check

The DO and GO primitives and the implied DO mechanism contain a test for whether the current maximum value for a VSTORE address is exceeded. Since within the interpreter relative VSTORE addressing is used, it is very unlikely that an address below the start of VSTORE could be generated.

15.2.8.3 Illegal Instruction Trap

The instruction transfer vector is 81 words long. The entries which have index values which do not represent a valid VM(M) instruction contain a transfer to the illegal instruction trap routine. This routine prints the VSTORE address of the illegal instruction and then transfers to the general error termination display routine.

15.3 Operation of the VM

The operation of MINT involves three steps: initiation of the Virtual Machine environment, loading of VSTORE, and Virtual-Machine instruction interpretation. The system is organized so that instruction interpretation is automatically started after VSTORE is loaded. One environment variable, MINT_HOME, is looked for when the VM is initialized. The path given by this variable is used to locate compiler or other text to be loaded by the VM.

15.3.1 Executing the Virtual Machine

The program which realizes the VM(M) Virtual Machine is an executable file named mint. This program is invoked by:

mint <options> <string>

where the options are:

-f "string" – string is a PDUMP-format or Portable-format file to be loaded into the VM
-i "string" – string is passed to MINT as first input line
-v – verbose signon: includes file load path and VM information

If no options are used, the file MCOMP.PDM is loaded from the MINT home directory given by the environment variable MINT_HOME.

15.3.2 Loading Virtual Memory

When the VM processor is executed, the first operation is to load Virtual Store with an executable MINT program. This executable program may be in one of two formats:

- Portable format
- Compressed core image (PDUMP) format.

PDUMP format loads more quickly, but is implementation dependent. Portable format can be loaded by any VM implementation. Portable format is described in Section 13.3. PDUMP-format files are created by means of the PDUMP primitive.

Normally, the system files are located in the subdirectory "mint3" in the source distribution. The Virtual machine loader uses the environment variable MINT_HOME to locate the MINT system files. This vari-

able should contain the path to the "mint3" directory in the distribution, or another directory where the MINT files have been put.

15.3.2.1 Compiler Source Input

When the compiler is loaded it responds with the message

MINT-3 Virtual Machine (32-bit Virtual Memory): Version V.L

The VM then loads VSTORE from a file. If the compiler is loaded it prints:

MINT-3 System: Version VV.LL. Created on: yymmdd

where VV.LL is the current version and level of the MINT compiler. At this point the system is ready for input from the primary input source. Input may be read from a file by means of the SI compiler directive. Input is directed back to the previous input stream on encountering an end-of-file.

15.3.2.2 Compiler Output

Listing output from the compiler, and any output from the user program using any of the string output functions described in Chapter 10 is normally directed to the primary output stream (e.g. a terminal window). This output may be redirected to a file by means of the SO directive. Output may be directed back to the primary stream by means of the OBREAK directive.

15.3.3 Upper/lower Case Convention

The MINT system distinguishes between upper and lower case in all contexts. All compiler primitives, directives, and functions are defined in upper case.

15.3.4 Use of CDUMP and PDUMP

The PDUMP primitive creates a PDUMP-format copy of the entire Virtual Store. This is useful for saving parts of a program that have already been debugged to avoid the need to recompile them. For example, suppose a program is made up of files prog_1, prog_2, and prog_3, and that all the routines in prog_1 and prog_2 have been checked out. They can be saved in compiled form by

mint

```
SI prog_1
SI prog_2
NOW PDUMP('prog.ok') !
```

The file name must be enclosed in quotes. Subsequently,

```
mint prog.ok
```

will load the compiler together with the compiled files prog_1 and prog_2. Execution will be restarted at the point of the PDUMP. At this point prog_3 may be compiled by:

```
SI prog_3
```

PDUMP is used in situations, as above, in which it is intended to resume processing at the current point after reloading the PDUMP image. However, it should be noted that the sequence:

```
NOW PDUMP('prog.ok') !
```

has the effect that a dictionary is pushed onto the dictionary stack by the NOW directive. This dictionary is popped from the stack by !. If it is intended to have the PDUMP'ed text start at some other address, the CDUMP directive should be used to avoid leaving a dictionary entry on the stack. For example, a new PDUMP format copy of the compiler is generated by:

```
CDUMP 'MCOMP.PDM' 32768
```

This text sets the start address to 32768 and uses PDUMP to write VSTORE to the file MCOMP.PDM.

15.3.5 Example Program Compilation and Execution

The following is an example use of the MINT VM diagnostic system. The diagnostic routine is an entirely standard MINT source program.

```
mint                    . line 1
LIST                    . line 2
DICT DIAGD:HDICT        . line 3
SI MINTDIAG             . line 4
BDIAG                   . line 5
```

This sequence carries out the following: line 1 starts the MINT VM which loads and starts the MINT compiler; line 2 sets source listing mode on, (if you are in a hurry or using a slow terminal you may want to skip

this line); line 3 introduces the dictionary DIAGD which is used by the diagnostic code; line 4 causes reading of the MINT diagnostic source text; line 5 causes MINT to start execution of the diagnostics. This should result in the output sequence shown in Section 14.2. In order to generate a new diagnostic file in portable format see the text in the file CDIAG.

15.3.6 The VM source Code

The C code for the MINT VM is quite simple. Changes to best match individual configuration needs can be made easily. We do not recommend changes to the code for the machine instruction loop or the machine instructions themselves. If such changes are made, the MINT diagnostics should be run to verify that no change has been made to the VM operation.

History of Corrections and Changes

1. The Mint Compiler

This Section provides a chronological record of changes to the compiler since the original 3.0 version.

1.1 14 January 2004: GETSTR Correction

The version has not been changed. The original version 3.0 was created on 020216, as shown in the signon line. This version has creation date 040113.

This is the first modification of the compiler since version 3 was developed. The change was to correct the omission of length checking in GETSTR. As described in the book, and as previously, GETSTR uses 34 word blocks for composition of the input strings and allocates a new block if the current one fills up. This occurs for each 132 characters of input. When GETSTR was rewritten to use the MINT 3 string format, the length test was omitted.

1.2 10 July 2004: Change to OPNL

Modified OPNL so that it updates the line count (RLNO) if PGLNGTH is non-zero. This eliminates the need for OPNLL as a compiler-internal function. This also improves the use of TITLE for pagination. The fact that OPNL did not update RLNO was noticed when adjusting the formatting of MTOC.

1.3 9 July 2006: Correction to ?, CURDIC, and introduction of MAXVS$

The directives ? and CURDIC did not return information about the active dictionary. They always returned information about the first directory in the directory list. This has been corrected. The VAR MAXVS$ has been introduced in order to provide the total size of VSTORE. VSTORE has been set to a large value (now 16M VSTORE words) for quite a while. The size is determined in the VM. The actual size acquired in the VM is

now set in the variable MAXVS$, which is in low VSTORE. The value is in units of 1024 words. A directive, VSTOREMAP, has been added to provide a convenient display of the layout and usage of VSTORE.

2. The C-coded Reference VM

This Section provides a chronological record of changes to the C-coded reference VM interpreter since the 1.1 version.

2.1 14 January 2004: Dynamic Operand Stack

The version is now 1.2.

Since the beginning, MINT has used fixed-size operand and link stacks. If it was found that the stack size was too small, the VM was recompiled with a larger stack. However, it was also true that the PDUMP format assumed the stack size in the current VM. Thus, if the stack size was changed, old PDUMP files would no longer load correctly.

The VM has been changed so that it dynamically allocates the stacks and the operand stack is automatically expanded if it fills up. The PDUMP format has been changed to contain the sizes of the stacks so that the PDUMP load routine will correctly load files with stacks of any size. The version level of the PDUMP format has been incremented to version 2 to distinguish this change. The PDUMP load routine will load both old and new version files.

2.2 15 August 2004: Correct addressing above 2^{31}.

The version is now 1.2.1

Corrected vm-c for VM addresses above 2^{31}. A number of variables were long which should have been unsigned long. In particular, PDUMP would not write correctly if the VSTORE end address was above 2^{31}. Experiments with Ulysses found this problem.

Also made improvements to diagnostic output. These include better formatting, handling of "segmentation faults," and correct value of program counter in dumps.

2.3 9 July 2006: Introduction of MAXVS$

The VM now sets the VAR MAXVS$ to the size of VSTORE (in units of 1024 VSTORE words.

2.4 19 September 2006: x86_64 and PDUMP Load

The version is now 1.2.3

The VM compiles and executes correctly on 64-bit systems, but in 32-bit mode. In addition, an architecture "endedness" test is now used to reorder integers during PDUMP load. This means that separate PDUMP files are not needed for big-end and little-end systems.

2.5 20 December 2015: Edits to the VM and one Manual update

These changes were made to allow compilation on Linux and Mac systems using clang and using the options -m32 and -m64. At present both -m32 and -m64 work correctly under Linux, but only -m32 (with #define x64_ set) functions under the current version (ver. 10.11.2) of the Mac OS. The code compiles using -m64 but it is not correct. No functional changes were made, but the version number was incremented to 1.2.4. Figure 13-10 was corrected. The text describing the Figure (for SLKP) was correct. The current VM code is in vm64-c.

Subject Index

!, 13, 14, 105
$, 134
$$, 146
∗, 191
−, 191
', 32, 33, 109
(, 109
(), 58, 92
), 109
*, 57, 61, 62, 191, 234
+, 14, 22, 57, 61, 62, 93, 156, 191, 197, 234
,, 109
-, 30, 57, 67, 191, 197, 234
., 38, 109
.mint_hist, 248
/, 57, 102, 191, 192, 234
//, 144
:, 23, 25, 109
;, 33, 134
;13, 23
;CR, 23
=, 77, 144
?, 108, 109
@, 55, 75, 109
[ ], 96
#, 31, 109
&, 32, 94, 109
-->, 199, 234
->, 73, 75, 177, 178, 247
<--, 199, 234
<=>, 184, 234
->, 234

Action
  Assignment, 17, 20
  Generative, 17
  Syntax, 16
ADD, 155
Addition, 155
ADIFF, 32, 60, 197
ADV, 73, 75, 177, 178, 234, 247
ADVCH, 100, 134
AND, 59, 196, 201, 234
ANSI, 7, 10, 21, 127
Apple-II, 1
ASUBS, 243
Attributes
  Class, 16
AUTO, 172, 239, 240
Auto-compiler, 235, 239, 240

B-tree, 123
BACK, 71, 83
Backus Naur Form, 143
Backward, 71
BASE, 139
Binding, 58, 60
BINLOC, 100
blank, 23
BLANKS, 100, 133
BLOCK, 28, 34, 48, 109
BNF, 143
Bound, 60
Brackets, 96
BTDEL, 100, 124
BTINIT, 100, 123
BTINSRT, 100, 124
BTREM, 100, 125

C, 12, 245
CAP, 128, 132, 136, 216, 219
carriage-return, 23
CDUMP, 87, 109, 229, 247, 251
Changes, 255
CHAR, 100, 134, 135

Character Address Pair, 128
Character case, 251
Character constant (#), 31
CHOOSE, 79, 204, 234
CLASS, 16, 106, 143
  CLASS, 17, 20
  DICT, 17, 20
  DIR, 18, 20
  FN, 18, 20
  ICON, 19, 20
  LAB, 19, 20
  MACRO, 19, 20
  PRIM, 18, 20
  VAR, 19, 20
Classes, 22
CLEARLST, 118
CLOSEF, 139, 223, 234
colon, 23
Colon(:), 25
Comma, 14, 62, 64, 66, 105, 148
Comment, 38
COMP, 241, 242
Compactness, 8
COMPILE, 100, 141
COMPL, 58, 191, 196, 201, 234
Conditional Execution, 80
Conditional transfer, 79
Constant, 19, 29
  Address, 31
  Character, 31
  Evaluated, 32
  Integer, 29
  String, 32
CONT$, 172
Control Transfer, 78, 204
Corrections, 255
CURCHS, 131, 132
CURCOL, 131–134
CURDIC, 108, 109

D$, 99, 104, 106, 109, 110
DATA, 20, 55, 109
Data-space, 9, 20, 25, 37, 55
DATE, 172
DECMP, 99, 100
Decompile, 106
DETACHED, 100, 114–116, 126
Diagnostics, 33, 236
DICMATCH, 73, 140–142, 216, 219, 234
DICT, 17, 44
Dictionary, 7, 9, 16, 17, 20, 21, 23, 25, 28, 61, 105, 121, 123, 141
  Entries, 7
DIFFER, 58, 191, 196, 201, 234
DIGIT, 100, 135, 156
DIR, 18, 103
Directive, 18, 103
Directives
  Table of, 109
DLOC, 9, 19, 26, 55, 67, 103, 133, 135
DO, 96, 98, 212, 230, 234
DREM, 172, 192
DUMP, 100, 117, 126
DUP, 76, 156, 184, 234

Editor, 151, 157
ELSE, 81
EMULATE, 229, 231, 234
Emulate, 85
End-of-String Action, 134
ENDBLOCK, 28, 34, 48, 109
ENTRY, 91, 93, 96, 103, 212, 234
EQ, 59, 201, 234
EQV, 25, 109
Escape, 33, 134
ESTOP, 34, 87, 229
EVAL, 12, 155
Evaluated Constant (&), 32
Exception
  Arithmetic, 191
Exchange, 76
Execution: Conditional, 80

EXIT, 12, 18, 34, 64, 81, 91, 93, 96, 98, 103, 110, 176, 188, 212, 234
EXOPT$, 172
Expression
  Address, 60
  Arithmetic, 57
  Boolean, 59
  Object Selection, 79
EXR, 87, 229, 230, 234

FAIL, 146
Failure Mechanism, 146
FALSE, 80, 205, 234
Fast-back, 143
Fixed Addresses, 172
FN, 18, 91
FNAME, 100, 135, 137
FORBTVAL, 100, 125
FORCHS, 118
FOREACH, 100, 117
FORGET, 22, 24, 28, 35, 109
Fortran, 148
Forward, 70
Free-space, 9, 34, 35, 96, 113, 114
Free-Space List, 114
FROM, 32, 60, 76, 197, 234
FULDIC, 108
Function, 18, 91
  Anonymous, 96
  Identified, 91
  Parameters, 91
  Reference, 92
  Table of, 100

GE, 59, 201, 234
GET, 20, 73, 177, 178, 205, 234, 247
GETCH, 73, 130, 216, 234
GETREC, 121
GETSTR, 100, 130
GETV, 20, 73, 177, 178, 205, 234, 247
GLKP, 98, 188, 234
GO, 78–80, 204, 234, 237
GT, 59, 80, 201, 234

Host, 87
Host machine, 11

ICL$, 29, 46, 109
ICON, 19, 29
Identifier, 7, 21
  Introduction, 22
  Local, 24
  local, 27
  Matching, 21
  Naming, 23
  Removal, 24
  Renaming, 24
  Setting, 25
IDLOC$, 172
Illegal, 234
Immediate Execution, 105
Immediate Setting, 26
Implicit reference, 212
INCH, 73, 130, 223, 234
INHEX, 133
ININT, 100, 133, 156
INSTRING, 96, 100, 132, 135
Instruction Execution Unit, 177
Instruction Set, 177
Intercept, 84
IOSBS, 241, 242
IPAR, 12, 15, 18, 26, 30–32, 57, 100, 103, 104, 155, 157, 240
IPAR-Expression, 103
ISO, 7, 10, 32, 127
Item Block, 116
Item List, 116
Iteration, 82

JOIN, 100, 114–116, 126

key compare, 123

LAB, 19, 23
Label, 19, 23
LASTDIC, 46
LCODE, 38, 67, 89, 109
LE, 59, 201, 234
LETTER, 100, 135
Linux, 12
LIST, 37–39, 67, 89, 109
List
  Free-Space, 114
  Item, 116
  Pointer, 113
  Record, 118
  Structure, 113
LISTDIC, 109
LISTDICS, 108
Literal, 29
Local, 27
Local Identifiers, 27
LOCK, 29, 46
LOCS, 37, 67, 89, 109
LOSE, 77, 82, 117, 184, 234
LP Register, 176
LT, 59, 201, 234
LV$, 108, 109, 243

M-TRAN, 143
Machine Independence, 4
MACRO, 19, 69
MASK, 58, 191, 196, 201, 234
MATCH, 73, 140–142, 216, 219, 234
MAXDLOC, 55
MAXDS$, 55, 172, 239
MAXPLOC, 55
MAXPS$, 55, 172, 239
MAXVS$, 56, 172
MINT, 1
  Analysis and Diagnostics, 152
  Functional Structure, 5
  Language Components, 15
  Size, 8
MINT_HOME, 250
MINTAUTO, 243
MINTCNF, 242
MINTCOMP, 242
MINTPRIM, 243
MINTSUBS, 242
MINTSYS, 241
MINUS, 30, 109
Miscellaneous Constructs, 84
MOVE, 99, 100, 126
Multiple Introductions, 22

N_d, 9
N_p, 9
NE, 59, 201, 234
NEG, 30, 57, 58, 191, 192, 234
NEXTCH, 100, 134
NEXTFREE, 35, 100, 115, 116
NO, 79, 204, 234
NOCHS, 133
node
  leaf, 125
  non-terminal, 125
NOLIST, 38, 39, 109
NOT, 59, 201, 234
NOW, 13, 14, 16, 105, 106, 109, 111
NULL, 100

OBJ, 108, 109
Object, 41
Object Selection Expressions, 79
OBREAK, 109, 137, 251
Obtain, 41
ODIR, 243
OPCH, 73, 136, 223, 224, 234, 237, 248
OPENF, 128, 132, 137, 140, 223, 234, 242, 248
Operand stack, 73, 76

Operator
  Address, 60, 197
  Arithmetic, 57, 191
  External, 127
  Logical, 59, 201
  Relational, 59, 201
  Shift, 57, 199
  Store, 178
  String, 127
OPFF, 100, 139
OPINT, 12, 14, 100, 137, 139, 155, 156
OPINTD, 100, 137
OPNL, 12, 100, 139, 155
OPTION, 147
OR, 59, 201, 234
Organic
  Programming, 6
  Structure, 5
Orthogonality, 5
Output, 135
OUTST, 32, 100, 137, 139
Overflow, 34

PAD, 138
PAGE, 38, 103, 109
Parameter
  Complex, 94
Parentheses, 14, 65, 148
parentheses, 58
PDUMP, 229, 234, 245, 247, 251
PEND, 144
PHRASE, 143
Phrase, 143, 147
Phrase Element, 144
Phrase Structure, 143
PLOC, 9, 19, 55, 67
PO, 240
Pointer
  Link, 176
  LP, 176
  SP, 176
Pop, 41
POPPEDUP, 100, 117, 120
POPUP, 100, 116, 126
POPUPREC, 100, 120
Portable Format, 173
Portable format, 10
Precedence, 60
Precedence Numbers, 64
Precision, 5
PREVDIC, 108, 109
PRIM, 18, 243
Primitive, 15, 18
Primitives
  External Interface, 223
  Table of, 234
Procedure-space, 9, 20, 25, 34, 37, 54
PROG, 20, 55, 109
Push, 41
PUSHD, 100, 116–118, 126
PUSHDREC, 100, 119
PUTCH, 73, 136, 216, 234

RCL$, 29, 46, 109
RD-ONLY, 46
READ, 100, 132, 156
Readability, 4
READINP, 100, 132, 134
readline, 248
Record List Pointer, 118
Recursion, 95
REF, 27, 104, 109
Register
  LP, 176
  SP, 176
RELREC, 121
RENAME, 22, 24, 103, 109
REPEAT, 82, 83
RESERVE, 56, 103, 105, 109, 110
Reset, 34
Reverse Polish, 41
Reverse-Polish, 10, 42, 60, 66
REVSTR, 235, 239

SAVBLOCK, 28, 29, 34, 49, 100
SEGIO, 129, 223, 224, 234
Self-Realization, 5
SETBLOCK, 28, 29, 49, 100
SETBSE, 100, 138
SETDIC, 45
SETOPP, 100, 138, 139
Shaw, G. B., 1
Shunt, 20
SI, 109, 131, 242, 251
SIUNIT, 131
Skip, 80
SLKP, 98, 188, 234
SO, 109, 137, 240, 251
SOUNIT, 137
SP Register, 176
Sperry Univac Series 1100, 1
Stack, 41
  Link, 98, 176
  Operand, 176
  Pointer, 176
Stack: Operand, 73, 76
START, 82, 83
States, 54
STOP, 87, 229, 234, 247
Storage Organization, 9
Store Operator, 75
String, 216
String Format, 127
String Match ($$), 146
Subject Index, 259
SUBS, 241, 242
Syntax Analysis, 143
SYSDAT$, 172
SYSID$, 172
SYSSTART, 240
System Generation, 243

T$, 84, 107, 109, 110, 230, 243
TEMP, 100
THEN, 81
TIME, 88, 229, 230, 234
TITLE, 38, 109
Top-down, 143
TRACE, 167
TRACTIVE, 230
Transformation, 4
TRAP, 84, 107, 166, 229, 230, 234
TRMIX, 167, 169
TRON, 230
TRUE, 80, 205, 234

Unconditional Transfer, 78
Undefined, 33
Uniformity, 8
UNION, 58, 191, 196, 201, 234
UNLOCK, 29, 46
Unmatched, 34

VAL, 73, 74, 79, 117, 177, 178, 234, 247
VAR, 19
Variable, 19
Virtual Machine, 1, 10
  Diagnostics, 236
  Loading, 174
Virtual Store, 171
VM, 11
VM(M), 1
VMDEBUG, 233, 234
VSTORE, 9, 31, 67, 73, 74, 84, 87, 98, 99, 107, 127, 157, 171, 177, 178, 188, 197, 223, 229, 246, 247, 249
VSTOREMAP, 56

WHILE, 71, 82, 83
WIDTH, 138

XOR, 59, 201, 234

YES, 79, 204, 234

Zero-Address Architecture, 42